UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

PEDRO PAULO PICCOLI FILHO

ANÁLISE EXPERIMENTAL DE CIRCUITOS, PACOTES E RAJADAS PARA IMPLEMENTAÇÕES DE REDES SOBREPOSTAS EM DATACENTERS CENTRADOS EM SERVIDORES

VITÓRIA
2013

PEDRO PAULO PICCOLI FILHO

ANÁLISE EXPERIMENTAL DE CIRCUITOS, PACOTES E RAJADAS PARA IMPLEMENTAÇÕES DE REDES SOBREPOSTAS EM DATACENTERS CENTRADOS EM SERVIDORES

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Engenharia Elétrica.
Orientador: Prof. Dr. Moisés R. N. Ribeiro.

VITÓRIA
2013

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Setorial Tecnológica,
Universidade Federal do Espírito Santo, ES, Brasil)

Piccoli Filho, Pedro Paulo, 1987-

P591a Análise experimental de circuitos, pacotes e rajadas para implementações de redes sobrepostas em Datacenters centrados em servidores / Pedro Paulo Piccoli Filho. – 2013.

109 f. : il.

Orientador: Moises Renato Nunes Ribeiro.

Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Telecomunicações. 2. Redes de computadores. 3. Telecomunicações - Equipamento e acessórios. 4. Sistemas de transmissão de dados. 5. Comutação de pacotes (Transmissão de dados). I. Ribeiro, Moises Renato Nunes. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 621.3

PEDRO PAULO PICCOLI FILHO

ANÁLISE EXPERIMENTAL DE CIRCUITOS, PACOTES E RAJADAS PARA IMPLEMENTAÇÕES DE REDES SOBREPOSTAS EM DATACENTERS CENTRADOS EM SERVIDORES

Dissertação submetida ao programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Mestre em Engenharia Elétrica.

Aprovada em 30 de Julho de 2013.

COMISSÃO EXAMINADORA

Prof. Dr. Moisés R. N. Ribeiro - Orientador
Universidade Federal do Espírito Santo

Prof. Dr. Divanilson Rodrigo de Sousa Campelo
Universidade Federal de Pernambuco

Prof. Dr. Magnos Martinello
Universidade Federal do Espírito Santo

“Uma longa viagem começa com um único passo.”
(Lao-Tsé)

A minha família e amigos.

Agradecimentos

Gostaria de agradecer a todos os professores e alunos do LabTel e em especial ao Professor Moisés R. N. Ribeiro por compartilhar seu conhecimento e experiência, aos amigos Flavio Rabello e Gilmar Luiz Vassoler. Também gostaria de agradecer a CAPES pelo apoio financeiro concedido, sem o qual provavelmente este trabalho não seria possível.

Sumário

O uso dos próprios servidores no encaminhamento/roteamento de pacotes propicia um ganho de flexibilidade e redução de custos e consumo de energia nas redes de datacenters. Todavia, a associação de uma nova funcionalidade aos servidores pode levar a uma redução em seu desempenho e por consequência influenciar no próprio desempenho da rede. Este trabalho investiga experimentalmente o processo básico de encaminhamento de pacotes por servidores. Serão analisados parâmetros como vazão, latência e perda de pacotes de forma a identificar o melhor modelo de implementação entre as alternativas de comutação de tráfego: circuitos, rajadas e pacotes. Como ferramenta de realização de experimentos foi utilizado o *Click Modular Router* assim como as modalidades nativas de encaminhamento do Kernel, enquanto para a coleta de parâmetros de desempenho utilizamos o conjunto *Oflops-NetFPGA*. Uma das principais conclusões é que com um eficiente método de classificação de pacotes, o modelo de rajadas, e encaminhamento via Kernel Linux se mostram como melhores opções na implementação das redes de datacenter centradas em servidores, alinhando uma maior vazão a um menor consumo da CPU.

Abstract

The use of servers themselves to forward/route packets provides higher flexibility, reduced costs, and lower energy consumption in data center networks. However, the association of a new task to the servers can lead to a reduction in their performance, and consequently, impair the performance of the network itself. This paper investigates experimentally the basic process of forwarding packets servers. In this paper, parameters such as throughput, latency and packet loss will be analyzed in order to identify the best implementation strategy among the following switching alternatives: Circuits, Bursts and packets. As a tool for conducting experiments Click Modular Router is employed; as well as native kernel forwarding/routing. For performance parameters monitoring *Oflops+NetFPGA* is employed. A key finding is that with an efficient method for packet classification, the model of bursts, and routing via Linux Kernel show themselves as better options in the implementation of network-centric datacenter servers, since they combine higher throughput with lower CPU consumption.

Sumário

Lista de Figuras

Lista de Siglas

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CWDM	Coarse Wavelength Division Multiplexing
DMA	Direct Memory Access
DWDM	Dense Wavelength Division Multiplexing
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
Gbps	Gigabit por Segundo
ICMP	Internet Control Message Protocol
IntServ	Integrated Services
IP	Internet Protocol
ITU-T	International Telecommunication Union
Mbps	Megabits por Segundo
OEO	Optical-Eletro-Optical
PrioSched	Priority Scheduling
QoS	Quality of Service
SDN	Software Defined Networking
TCP	Transmission Control Protocol
TDM	Time Division Multiplexing
TTL	Time To Live
UDP	User Data Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

Capítulo 1: Introdução

No presente capítulo serão citadas as motivações para o desenvolvimento deste trabalho, tais como os novos desafios encontrados com a grande expansão de serviços e aplicações no campo das telecomunicações e em especial nas redes de computadores, citando como exemplo as redes de *datacenter*. Algo que será também aqui tratado é a fundamental importância de ferramentas que viabilizem a pesquisa e desenvolvimento de novas tecnologias que por sua vez possam atender às necessidades atuais e futuras em se tratando de redes de telecomunicações.

1.1 Cenário de redes atual e futuro

Atualmente, com a disponibilidade de maior banda e acesso à Internet, está sendo ofertada uma variedade cada vez maior de serviços. Por consequência, tal fato impõe as redes de telecomunicações demandas cada vez maiores, como exemplo troca de dados em tempo real, serviços de armazenamento de dados e de buscas. Tais demandas não se encontram apenas restritas a grandes consumidores, mas também à simples usuários, que cada vez mais passam a utilizar mais e mais serviços. Fato esse que tem levado as redes de telecomunicações a um comportamento diferente do que se imaginava há alguns anos [1][2][3].

Tendo em vista o crescimento da Internet e de seus serviços, tem-se observado também que nos últimos anos, grandes investimentos foram feitos em centros de dados, *datacenters*, locais onde são concentrados equipamentos, como servidores, responsáveis pelo processamento e armazenamento de dados de uma empresa ou organização, massificando apoio aos serviços em nuvem. Serviços estes que constituem na possibilidade de uso de recursos de infraestrutura de vários servidores compartilhados, interligados e disponíveis por meio da Internet para tarefas tais como cálculos e armazenamento de dados, por empresas como eBay, Facebook, Google, Microsoft, Yahoo, dentre outras. Como consequencia, em tempos recentes, cada vez mais *datacenters* vem sendo construídos de forma a prover o grande crescimento dos serviços de aplicação online, como e-mail, buscas, e jogos, além do fato de os *datacenters* também possuírem a tarefa de desempenhar serviços de usuários, como

transferências e armazenagem de arquivo, realizando uma série de processamento interno. Com isso nota-se que os *datacenters* passaram a constituir um elemento fundamental para prover os recursos atualmente requeridos por diversos seguimentos de serviços e destinados a qualquer tipo de usuário [4].

Em se tratando de redes de *datacenter* existem algumas arquiteturas propostas, dentre as quais tem-se como exemplo: arquiteturas baseadas exclusivamente em *switches*, onde o encaminhamento de pacotes é implementado exclusivamente com switches. Esta categoria de redes de dados de *datacenter* tradicionais baseadas em *switches* podem ser organizadas segundo uma topologia hierarquica em árvore, tal como *Fat-Tree* [16] ou de forma aleatória como *Jellyfish* [17]. Uma segunda outra categoria, denominada por arquitetura híbrida centrada em servidores, incluem propostas de arquiteturas em que os pacotes são enviados usando uma combinação de switches e servidores, tais como Dcell [18] e Bcube [19]. Esta combinação explora a possibilidade de que cada servidor possa desempenhar um duplo papel: a execução de aplicações regulares e também o encaminhamento de tráfego entre servidores. Cada servidor é ligado a alguns outros servidores para formar uma topologia de *datacenter* totalmente interconectada por via de ligações entre servidores com ou sem a participação de simples *switches*.

As topologias baseadas em servidores ao contrário das baseadas em *switches* apresentam maior flexibilidade. De fato o servidor apresenta uma maior flexibilidade de configuração por via de *software* quando em comparação aos *switches*, algo que vem ganhando espaço com a idéia de redes definidas por software (SDN), a qual está diretamente relacionada ao presente trabalho [5].

1.2 Necessidade de conformação de tráfego e diferentes formas de comutação

Algo que se tem discutido ultimamente é se as formas tradicionais de comutação de dados, como comutação de pacotes e estabelecimento de circuitos, continuam sendo ainda a maneira mais eficaz para se aplicar a todos os tipos de redes e de fluxos de dados existentes.

A ideia por trás da conformação de tráfego é a de tornar possível controlar o tráfego saindo de uma interface a fim de que seu fluxo possa se adequar a velocidade da interface de destino, além de se adequar as demais políticas contratadas junto ao provedor e limitações da rede, por exemplo. De forma primária a conformação se aplica no controle do uso de banda, no estabelecimento das políticas de tráfego e na regulamentação do fluxo de dados buscando evitar congestionamentos e por consequência evitando a perda de pacotes [6].

1.3 Necessidade de plataforma aberta para pesquisa

Quando se pensa em equipamentos de rede de telecomunicações ou computadores nos deparamos sempre com equipamentos que são verdadeiras *"caixas pretas"*. De fato isso ocorre, pois embora os fabricantes sigam as normas do setor, não é de conhecimento, por parte do cliente ou pesquisadores, a arquitetura interna do equipamento. Outro ponto negativo é que dificilmente é possível adaptar do equipamento segundo as necessidades do cliente, e mesmo quando existe tal possibilidade a mesma se apresenta extremamente custosa.

De forma a facilitar o desenvolvimento e teste de novas tecnologias ou arquiteturas, que em muitos casos não são possíveis de se implementar em equipamentos comumente encontrados no mercado, em geral busca-se por sistemas de plataforma aberta. A opção pelos sistemas de plataforma aberta se dá pelo fato da mesma tornar possível o acesso total ou ao menos parcial ao código do software ou ao hardware. Através do uso de plataformas abertas é possível a simulação e modelagem de novas funcionalidades mais bem adaptadas a eventos de particular interesse do desenvolvedor.

Dentre os sistemas de plataforma aberta para desenvolvimento tem-se a API *Click Modular Router* [7], que foi uma utilizada no desenvolvimento da pesquisa tratada por este trabalho. A mesma será na sequencia comentada de forma breve e em um capítulo posterior a descrita de forma mais detalhada com exemplos de utilização.

1.3.1 API *Click Modular Router*

Desenvolvida por Eddie Kohler em meados de 2001, a API *Click Modular Router* [7][8][9] é uma ferramenta voltada a implementação de arquiteturas de rede já consolidadas ou em fase de desenvolvimento e teste. O *Click Modular Router* tem sua operação baseada em blocos funcionais chamados *elements*, os quais desempenham funções específicas e quando associados possuem a capacidade de desempenhar funcionalidades de equipamentos de rede como switch, roteadores, firewalls, dentre outros. Em seu funcionamento se mostra como uma ferramenta flexível, pois se baseando em blocos funcionais, torna possível ao desenvolvedor associá-los sem muitas restrições [8][9].

1.3.2 Redes Definidas por *Software*

Um conceito que tem ganhado uma grande expressão no cenário de redes é o das redes definidas por software (SDN)[10]. Definida como uma forma de se construir políticas de controle do tráfego de dados de forma distribuída com um elemento de gerenciamento centralizado. Em SDN as consultas as tabelas de encaminhamento continuam sendo atividades desempenhadas pelo hardware, entretanto a decisão de como cada pacote deve ser tratado é delegada a uma outra estrutura, centralizada e flexível, do ponto de vista que pode ser programada, sendo possível a agregação de novas funcionalidades. Devido à sua flexibilidade, o conceito de SDN abre a possibilidade de se desenvolver novas aplicações que controlem elementos de comutação de forma totalmente nova e impensada no passado, criando controladores de rede com lógicas de monitoramento de tráfego mais sofisticadas e com uma maior abstração do ponto de vista dos usuários da rede.

Uma rede SDN é caracterizada pela existência de um plano de controle, baseado em software, que age de forma a controlar e inspecionar o mecanismo de encaminhamento dos elementos que agem na comutação dos pacotes [10].

Como um dos principais representantes de SDN, tem-se a arquitetura *OpenFlow* [11]. A filosofia por trás da arquitetura *OpenFlow* surgiu com a ideia de tornar as redes de telecomunicações/computadores mais virtualmente flexíveis possível, embora ainda que não o sejam fisicamente, devida a própria topologia física. Para tanto a proposta adotada foi a de

criar elementos de comunicação (equipamentos) que pudessem ora realizar funcionalidades extremamente simples, como uma comutação, e ora pudessem suportar atribuições mais sofisticadas. Como funções mais sofisticadas, pode-se pensar na execução de algum novo algoritmo de roteamento ainda em desenvolvimento ou teste. A ideia da SDN é associar um tratamento simples para alguns tipos de pacotes, ao mesmo tempo em que realiza um tratamento mais refinados para outros no mesmo equipamento de comutação. Portanto o equipamento de comutação SDN pode se comportar como um simples switch ao mesmo tempo em que atua como roteador ou firewall. A fim de que tal filosofia pudesse ser realizável houve a necessidade de se criar uma separação entre a infraestrutura de dados e a de controle, criando assim um plano de controle e plano de dados mais simples. Dada à simplificação do plano de dados, o plano de controle passou a possuir atribuições mais complexas. Criando-se assim um elemento de controle centralizado com atribuições de realizar tarefas de controle e supervisão dos demais componentes. Como forma de garantir o bom funcionamento da rede, todo o processo de controle e supervisão do controlador é realizado via um canal de comunicação seguro entre o controlador e os elementos de comutação ou *switches*, Figura 1.1.

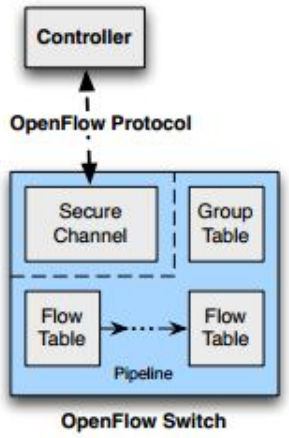

Figura 1.1 - Comunicação Controlador-Switch OpenFlow via canal seguro [12].

Embora o controlador seja centralizado, pois toma ações relativas ao comportamento de um ou mais comutadores, a lógica de atuação do controlador dentro da infraestrutura *OpenFlow* é vista como distribuída [12].

A grande vantagem nativa do *OpenFlow*, sendo característica do paradigma SDN, é a flexibilidade, pois o mesmo pode ser programado, criando a possibilidade de que o mesmo possa se comportar da forma que for mais conveniente ao projetista. Tal funcionalidade

viabiliza o seu uso no desenvolvimento e teste de novos protocolos de redes [13]. A ideia é que simultaneamente um switch *OpenFlow* se comporta como um simples *Switch Ethernet*, para determinados quadros, também pode realizar tarefas diferentes agregando uma maior inteligência a rede a outros tipos de dados. Tal característica possibilita que uma rede *OpenFlow* possa dar suporte ao mesmo tempo a diferentes protocolos de rede e até mesmo protocolos em via de desenvolvimento. O seu funcionamento dá-se via instalação de regras nos *switches* pelo controlador, que associadas às identificações dos campos do cabeçalho dos pacotes, possibilitam um tratamento diferenciado a cada pacote [14].

1.4 Redes de *datacenter*

Há uma tendência crescente para a migração de aplicações e armazenamento em centros de disseminação de dados através da Internet. *Datacenters* contêm atualmente dezenas de milhares de computadores requerendo significativa banda agregada de dados. De maneira que as redes de *datacenters*, sendo fundamentais para o transporte interno de grandes volumes de dados, tem-se tornado um assunto de muita discussão e pesquisa [15][16][17][18].

Em se tratando de redes de *datacenters* existem as topologias clássicas centradas em switches e outras centradas em servidores, que recentemente tem atraído uma atenção especial devido à sua flexibilidade associada ao crescente desenvolvimento de pesquisas em redes definidas por *software*. As redes centradas em switches se destacam quanto a topologia mais simplificada, devido a possibilidade de maior número de interfaces destinadas a realizar a interconexão com demais switches ou computadores, outra vantagem está no hardware dedicado desses equipamentos. Quanto as redes centradas em servidores, sua grande vantagem encontra-se na flexibilidade, uma vez que os servidores pondem ser programados de forma a se comportar diferentemente para cada pacote ou fluxo de dados recebido.

Na sequência serão exemplificadas quatro topologias de redes de *datacenter*, duas centradas em switches e duas centradas em servidores. As últimas se tornam importantes neste trabalho, pois o mesmo trata de aspectos relevantes encontrados nas redes centradas em servidores.

1.4.1 *Fat-Tree*

Fat-Tree [15] é uma topologia baseada em switch, onde os mesmos são os responsáveis pelo encaminhamento de dados. A *Fat-Tree* possui como principal objetivo, a redução de custo associada à manutenção da capacidade de utilização da banda entre os servidores. A topologia é construída na forma de árvores com várias camadas, onde os servidores se encontram na base interconectados via switches de forma a reduzir o custo da arquitetura *Fat-Tree*. Os switches por sua vez se ligam a outros switches formando as demais camadas da rede. Dessa forma tal estrutura permite que todos os servidores possam se comunicar, assim como ilustrado na Figura 1.2.

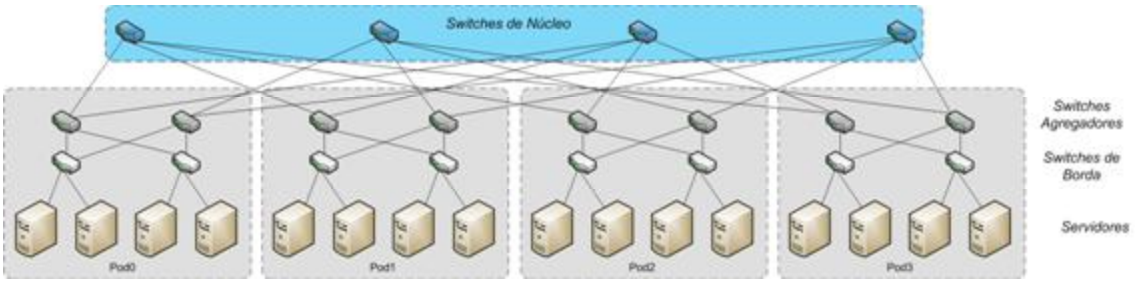

Figura 1.2 - Topologia *Fat Tree*.

Na arquitetura *Fat-Tree* a conexão entre os elementos segue a seguinte regra de construção: uma *k*-ésima *Fat-Tree* é composta por *k pods*, que são estrutura composta pelos servidores e switches ou comutadores diretamente ligados aos mesmos, Figura 1.2, e $k^2/4$ comutadores de núcleo com *k* interfaces cada. Um *pod* é sub-dividido em duas camadas, uma de borda e outra de agregação, cada uma com *k/2* comutadores, além de *k* servidores. Para um comutador de borda com *k* interfaces, *k/2* intercafes são utilizadas para conectar o comutador aos servidores e as *k/2* interfaces restantes ligam o comutador à comutadores da camada de agregação.

A fim de atingir uma alta capacidade de comunicação entre servidores, distribuindo da forma mais uniforme possível o tráfego de saída de um *pod* entre os comutadores de núcleo, foi proposto o uso de tabelas de roteamento em dois níveis de forma a espalhar o trávego baseando-se nos *bits* menos sgnificativos do endereço IP de destino. De forma simplificada, o endereçamento em um *pod* segue o seguinte formato: *X.pod.comutador.1*, onde *pod* indica o

número de identificação do *pod*, variando de 0 a *k-1*, enquanto *comutador* identifica a posição do comutador no *pod*, também de 0 a *k-1*, e *X* indica o bloco de endereço IP, do tipo *X.0.0.0/8*, alocado para uso dentro a rede *Fat-Tree*. Quanto aos comutadores de núcleo, o endereçamento segue a seguinte regra: *Y.k.j.i*, onde *i* e *j* identificam os $k^2/4$ comutadores, o *id* identifica a posição do servidor na sub-rede, variando de *2* a *k/2 + 1*.

1.4.2 *Jellyfish*

Jellyfish [16], assim como a *Fat-Tree*, é uma topologia baseada em switches, construída tomando por base camadas de switches que ligam outros switches e servidores. Em *Jellyfish*, se cada switch possui *k* portas, *r* destas portas são destinadas para ligarem o switch a outros switches de forma aleatória, já as *k-r* portas restantes são usadas para ligar servidores. No *Jellyfish* não há uma hierarquia, as conexões são feitas de forma aleatórias.

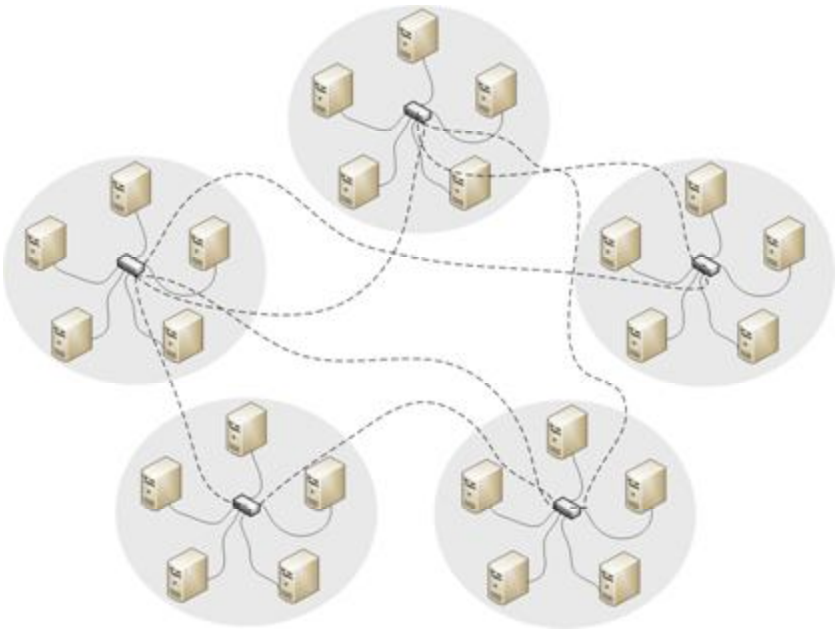

Figura 1.3 – *Jellyfish*.

A proposta da arquitetura de *datacenter Jellyfish* procura tratar da expansão da rede de forma incremental. Em arquiteturas hierarquizadas como o *Fat-Tree* e *BCube*, a inserção de novos servidores, se não prevista inicialmente, acaba se tornando custosa devido a própria

hierarquia nas quais as arquiteturas são estruturadas, necessitando da inserção de novos switches e servidores para realisar conexão com as camadas superiores. Caso o aumento do número de servidores seja feito sem levar em consideração as características topológicas de cada um das arquiteturas, a alteração pode gerar um desbalanceamento da estrutura do *datacenter* levando a uma perda de desempenho.

De forma a tornar a expansão dos *datacenter* mais simples e ao mesmo tempo manter suas propriedades foi idealizada a arquitetura *Jellyfish*. No *Jellyfish* as conexões são feitas de forma aleatórias entre os comutadores, levando a uma distribuição uniforme, não hierarquizada e de simples expansão. Além de simplicidade quanto à expansão, o *Jellyfish* possui a vantagem de possibilitar o uso de diferentes comutadores em uma mesma rede, algo que não é possível em outras arquiteturas de *datacenters*. Em comparação com o *Fat-Tree*, o *Jellyfish* usando os mesmos equipamentos pode suportar 25% mais servidores, destacando desta forma tanto na simplicidade quanto na expansão.

1.4.3 DCell

A topologia do DCell [17] se baseia em módulos, sendo portanto montado de forma recursivamente. No DCell são utilizados tanto servidores quanto mini-switches para o encaminhamento de pacotes. O principal módulo desta topologia é $DCell_0$, célula DCell nível 0, que é formado por um switch e *n* servidores. Cada $DCell_0$ é constituida por um conjunto de *n* servidores conectados a um mini-switche, que tem a função de prover a comunicação interna entre os servidores. A conexão entre $DCell_0$ diferentes é feita via ligação entre o servidor pertencente a uma $DCell_0$ e o servidor de outra.

O $DCell_1$ é construído através das interconexões de $n + 1$ células $DCell_0$, onde *n* denota o número de servidores internos da $DCell_0$. Um $DCell_0$ está ligado a todas as demais células $DCell_0$ por um elo de ligação entre as diferentes células DCell de nível 0, como ilustrado em Figura 1.4, onde temos uma célula DCell de nível 1, $DCell_1$, sendo formada por 5 $DCell_0$s, identificadas por: $DCell_0 0$, $DCell_0 1$, $DCell_0 2$, $DCell_0 3$e $DCell_0 4$.

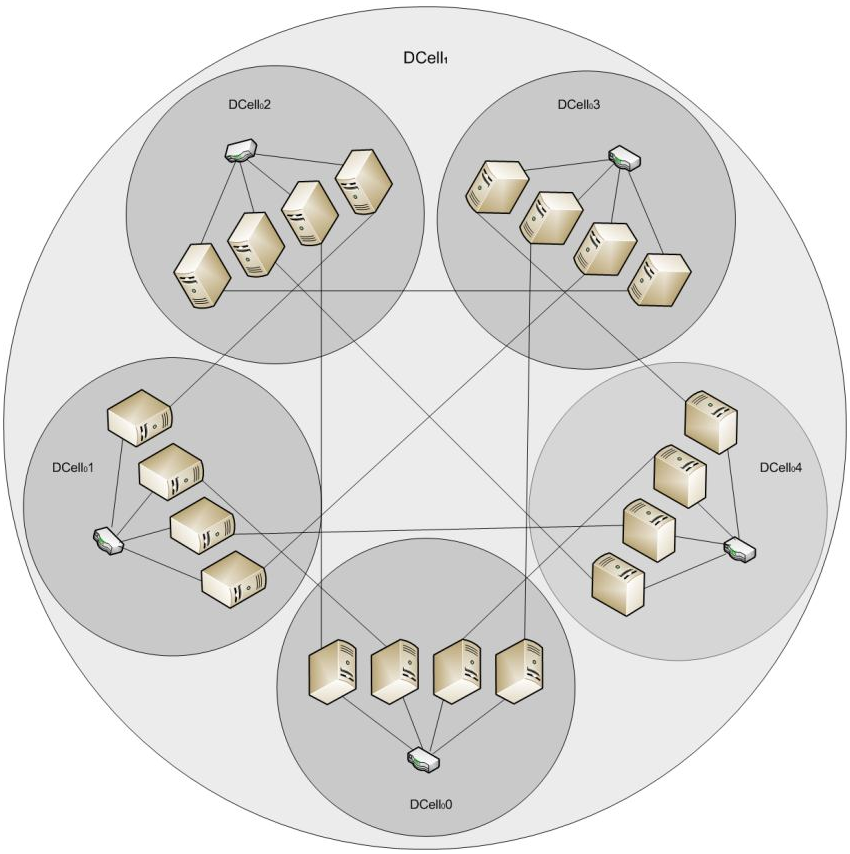

Figura 1.4 - Topologia *DCell*.

Nota-se que no DCell, diferentemente do BCube, os *switches* estão ligados apenas aos servidores na mesma Dcell, sendo a conexão entre diferentes redes DCell sendo sempre feito por via dos servidores.

Como características do DCell tem-se a capacidade de fácil expanção e grande tolerancia a falhas quando comparado com as arquiteturas baseadas em árvores, como *Fat-Tree*, além de proporcionar uma alta capacidade de banda de dados.

1.4.4 BCube

Dentre as propostas de construção de *datacenters*, têm-se os modulares construídos no interior de contêineres (MDC – *Modular datacenter*). Embora nos MDCs utilizem de

pequenos switches nas interconexões, eles seguem o modelo de *datacenters* centrados em servidores, onde toda a inteligência da rede encontra-se nos servidores. Com a construção de módulos de *datacenters* no interior dos contêineres a estrutura se torna modular, tornando possível um menor tempo de na montagem dos módulos e sua implantação com associação de vários módulos, também permite uma maior mobilidade, além reduzir os custos quanto ao resfriamento, já que os módulos são totalmente lacrados pela estrutura do contêiner. Entretanto os MDCs apresentam uma maior dificuldade em se tratando de manutenção quando comparados a outros modelos de *datacenters*, pois a estrutura dificulta o acesso aos equipamentos que constituem cada um dos módulos. Sendo assim, o MDC é precisa de uma alta tolerância a falhas. Dentre os *datacenters* do tipo MDC, o principal exemplo é o BCube [18]. O BCube foi idealizado de modo a possuir uma lenta queda de desempenho quando submetido a situação de falhas em seus servidores. Além do desempenho, o BCube também foi idealizado a fim de permitir o suporte as comunicações: um-para-um, um-para-muitos, um-para-todos e todos-para-todos. Como forma de aumentar a tolerância a falhas e melhorar o balanceamento do tráfego, o BCube possibilita a criação de múltiplos caminhos paralelos e curtos entre os nós da rede.

Quanto a arquitetura, o BCube, Figura 1.5, possui dois tipos de elementos: servidores com múltiplas interfaces de redes e pequenos switches. O Bcube é uma proposta de arquitetura de rede multi-nível hibrida, embora centrada em servidores, para *datacenter* com a seguinte característica: os servidores, além dos *switches*, também fazem parte da infra-estrutura de rede, ou seja, eles também atuam encaminhando pacotes em favor de outros servidores. A estrutura do BCube se baseia em recursividade a partir de uma base chamada nível zero ou $BCube_0$, constituído por um conjunto de n servidores conectados a um switch e n portas. No nível 0, o $BCube_0$ é constituído por n servidores, que se conectam por via de um simples switch. Já o $BCube_k$ é formado pela associação de n $BCube_{k-1}$ conectados com n^k comutadores de n interfaces cada. Os n $Bcube_{k-1}$ são enumerados de 0 a $n-1$ e os servidores em cada $Bcube_{k-1}$ são numerados de 0 a n^k-1. A conexão dos módulos segue a seguinte lógica: a interface k do i-ésimo servidor localizado no j-ésimo $Bcube_{k-1}$ é conectada à j-ésima interface do i-ésimo comutador de nível k.

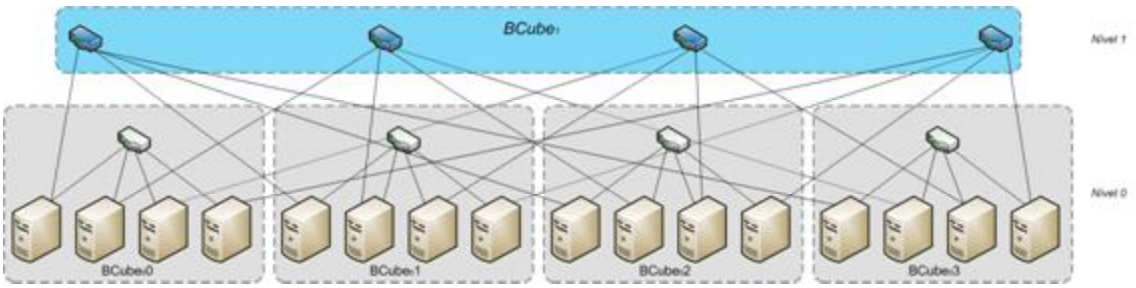

Figura 1.5 - Topologia *BCube*.

Como já citado anteriormente, a topologia BCube foi proposta para ser utilizada na construção de *datacenters* Modulares (MDC), *datacenters* construídos dentro *containers* de transporte, permitindo uma instalação mais simples, além de facilitar os procedimentos físicos de migração quando em comparação com *datacenters* regulares. A arquitetura de rede Bcube, apesar de ainda fazer uso de simples switches, toma uma abordagem centrada em servidores o que possibilita a inserção de maior inteligência e flexibilidade na rede quando comparada as centradas em *switches*.

1.5 Circuitos, pacotes e rajadas

As topologias de redes de *datacenter* físicas apresentadas anteriormente podem propiciar diferentes modalidades de comunicação entre os servidores dependendo das características exigidas na comunicação e dos elementos de rede envolvidos, como tempo de permanência, vazão, *Jitter*, dentre outras. Com relação aos modelos tradicionalmente mais conhecidos tem-se, comutação por circuitos, pacotes e um terceiro método denominado encaminhamento por rajadas. Predominantemente as redes de computadores baseiam-se na comunicação de pacotes. Entretanto tal modelo pode não ser eficaz para todos os tipos de demandas, devido às características do fluxo de dados. Portanto neste trabalho propomos investigar o consumo de CPU dos elementos envolvidos no estabelecimento de circuitos e rajadas, tendo como aplicação as redes de *datacenters* centradas em servidores.

1.5.1 Comutação de Circuitos

Em resumo, o chaveamento de circuito ou c*ircuit switching*, Figura 1.6, possui três fases distintas, que são: estabelecimento do circuito, com a reserva do meio de comunicação e a identificação dos integrantes como origem, destino e nós intermediários. Com o fim da primeira etapa, inicia-se propriamente a transmissão de dados e após o seu fim, ou prazo de tempo associado ao uso do meio, inicia-se o encerramento da comunicação e desalocação do canal, passando o mesmo a estar disponível a demais usuários que queiram utilizá-lo.

A primeira etapa, a de estabelecimento do circuito, onde é usado algum tipo de protocolo de "configuração" distribuído ou centralizado a fim de encontrar o caminho e reservar os recursos, que podem ser, por exemplo, faixas de frequência, intervalo de tempo e comprimento de onda. Em uma comunicação por circuito o tempo T necessário para o estabelecimento do circuito é de no mínimo $2P + \Delta$, onde P é o tempo de propagação entre uma ponta e outra do circuito e Δ o tempo de atraso imposto pelo processamento ao longo do mesmo. A comunicação por circuito se caracteriza por um modelo de comunicação orientado a conexão [19][20]. Os serviços orientados a conexão garantem qualidade de serviço, pois o nó de origem só envia dados quando tem garantia de que existe um caminho até o destinatário que tem capacidade de atender a exigências da transmissão de dados, além da própria aceitação do destinatário. Antes que dois sistemas, computadores, por exemplo, possam trocar dados, há a necessidade do estabelecimento de uma conexão entre os mesmos. Para tanto, em um primeiro momento é identificam-se os nós que farão parte da conexão. Na sequencia é realizada a configuração e alocação dos recursos necessários ao longo de todo o caminho, associada à sinalização entre os sistemas finais. Ao termino da primeira etapa, inicia-se a transferência de dados e por fim ocorre a terceira etapa, que é a encerramento do circuito ou termino da conexão, Figura 1.6.

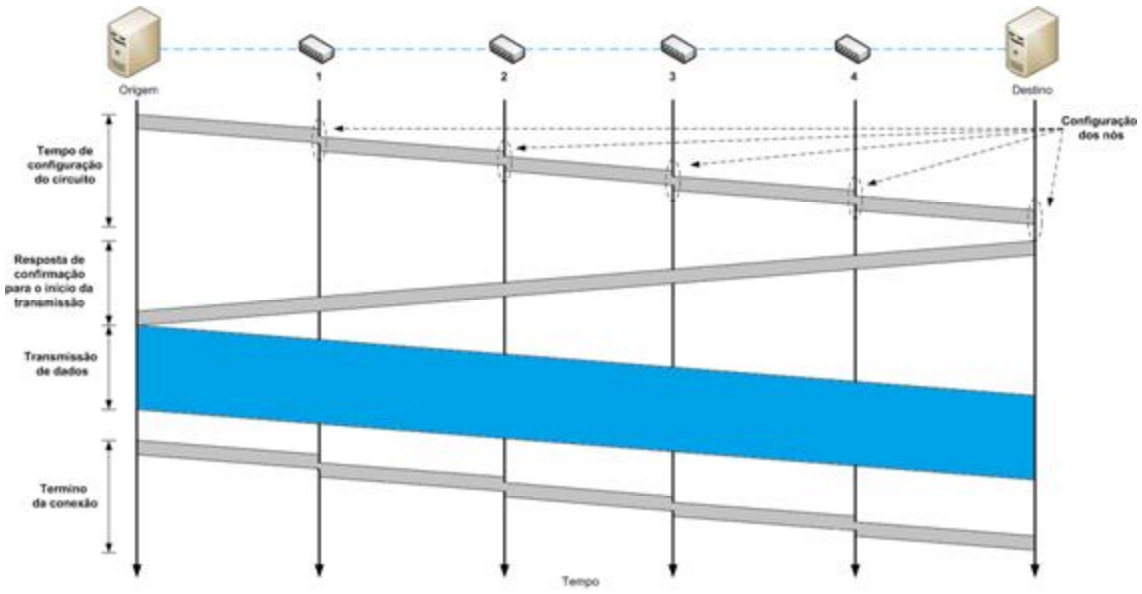

Figura 1.6 - Etapas de estabelecimento de um circuito.

Devido à alocação dos recursos desde o ato de estabelecimento da comunicação por circuito até o momento de seu fim, a mesma só se mostra eficaz se o tráfego seguir um modelo contínuo ou com pouca variação e de longo período de tempo. Tal comportamento não se assemelha ao apresentado pela internet, tendendo a mesma a se aproximar de um modelo de rajadas (*bursty*) em escala de tempo.

1.5.2 Comutação de Pacotes

Em se tratando de *packet switching*, ou comutação por pacotes, Figura 1.7, cada pacote passa a ser independente dos demais. Podendo cada um dos mesmos seguir rotas diversas, uma vez que cada pacote possui seu próprio cabeçalho com endereços de origem e destino. Essa característica possibilita uma capacidade de melhor explorar a capacidade da rede quanto as possíveis rotas. Para o caso do *packet switching* os pacotes passam a ter tamanhos limitados entre valores máximos e mínimos, S_{max} e S_{min} respectivamente. Em geral o tamanho está ligado ao tipo de dado transportado e protocolo utilizado no mesmo, como Ethernet, ATM, etc. Com a fixação de tamanho, o fluxo de dados passa a ser fragmentado em pacotes de tamanho variável entre S_{max} e S_{min} [19][20][21].

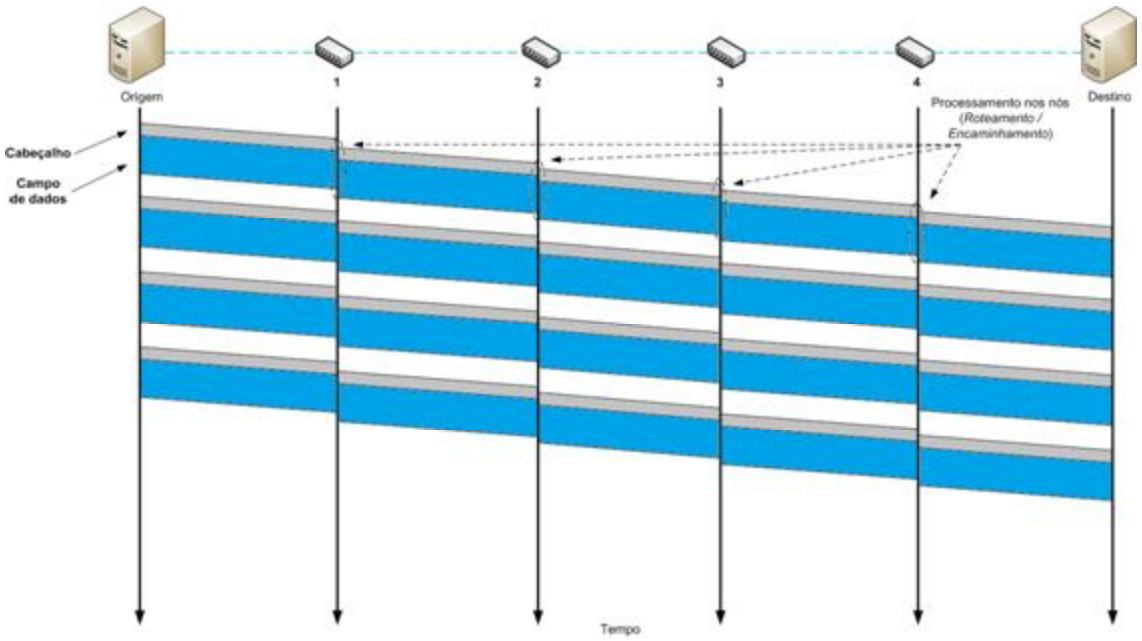

Figura 1.7 - Comutação de pacotes.

A comutação de pacotes se apresenta mais comumente associada à forma de encaminhamento conhecida como *store-and-forward*. Nessa forma de encaminhamento cada quadro precisa ser analisado, ou seja, desmontado, processado e remontado em cada um dos nós por onde passar. Para que toda essa manipulação seja possível, há a necessidade de *buffers* com capacidade suficiente de acumulação em cada um dos nós intermediários. O uso de tal artifício acaba por elevar a latência com o aumento do número de nós intermediários. Embora o tamanho do *buffer* também esteja relacionado ao tamanho do pacote, além de outros aspectos de projeto, sendo quanto menor pacote, menor será o tamanho do *buffer* necessário. A redução do pacote não leva a redução de seu cabeçalho, gerando uma alta relação cabeçalhos por dados efetivos, reduzindo a capacidade da rede.

1.5.2.1 Roteamento e Encaminhamento de Dados

O processo de roteamento é tido no âmbito geral da rede como a forma de determinar os caminhos fim-a-fim que os pacotes devem percorrer desde a fonte até o destino [21].

Na Internet o roteamento é a principal forma utilizada para a entrega de pacotes de dados entre hosts. Em geral, a métrica de comutação mais utilizada é a *hop-by-hop*. Cada roteador que recebe um pacote de dados inspeciona o seu cabeçalho, e com o endereço de destino do cabeçalho IP, por exemplo, identifica o próximo salto em direção ao destino. Tal processo vai se repetindo até que o pacote atinja seu destino ou por algum motivo o mesmo venha a ser descartado. Entretanto, para que seu funcionamento seja possível, são necessários dois elementos: tabelas de roteamento e protocolos de roteamento.

Tabelas de roteamento de forma básica são registros que relacionam endereços de destino a número de saltos até o mesmo, podendo ocorrer muitas outras informações também associadas às tabelas.

Os Protocolos de roteamento determinam o conteúdo das tabelas de roteamento e ditam o algoritmo usado no processo de roteamento. Definem a forma como a tabela é montada e quais informações serão associadas a ela. Além de especificar como o conjunto de informações será usado no processo de encaminhamento do pacote ao próximo nó. Portanto o processo de roteamento se refere à escolha de um ou mais caminhos, ou rotas, a serem seguidos pelo pacote no âmbito da rede, enquanto o encaminhamento está focado na atividade local de entregar o pacote ao próximo nó da rota.

Como principais algoritmos de roteamento atualmente em uso têm-se, o baseado em Vetor de Distancia *(Distance-Vector Routing Protocols)* e o baseado no Estado de Enlace *(Link State Routing Protocols)*[19][21].

1.5.3 Comutação por Rajadas

O *Burst Switching* ou comutação por rajadas, Figura 1.8, se apresenta como uma proposta alternativa de comutação de redes. Tem como especificação a unidade de transmissão o encadeamento de pacotes que apresentam características semelhantes, sendo denominado por rajada. As rajadas apresentam granularidade intermediária entre a comutação por circuito e por pacotes, possuindo desta forma características tanto do *packet switching* quanto do *circuit switching*, tomando para si as melhores características de cada um [22].

De forma simples, redes baseadas em *Optical Burst Switch* possuem dois tipos básicos de componentes: um de borda ou agregador, cuja função é de classificar e montar os blocos em rajadas, além de enviar a rajada de dados ao segundo componente; outro de núcleo, responsável por rotear ou encaminhar o *burst* até um novo componente de núcleo ou de borda, onde, no caso do último, a rajada será desmontada e entregue a um usuário ou outra infraestrutura que não se baseie em fluxo de rajadas.

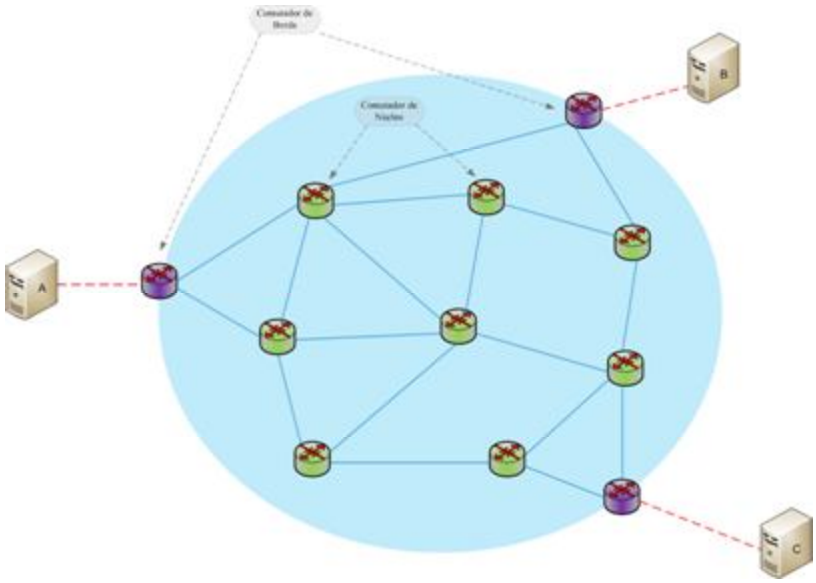

Figura 1.8 - Arquitetura básica de uma rede OBS.

No *Optical Burst Switching* cada rajada é precedida por um pacote de configuração responsável por fornecer aos componentes de núcleo informações sobre o caminho a ser percorrido pelas rajadas, Figura 1.9. Esse mesmo pacote também transporta informações

referentes à reserva de banda de dados necessária dependendo da disponibilidade de cada nó. A forma como é transmitido e manipulado o conjunto, pacote de configuração e rajada, possibilita a implementação de diversas arquiteturas de OBS, variando o modo como é feito todo o controle e reserva de banda/canal de dados. Em muitas arquiteturas há uma separação entre plano de dados e de controle, tendo o último a capacidade de estar associado a mais de um plano de dados. De forma que, enquanto no plano de dados são enviadas rajadas com centenas ou milhares de pacotes, no plano de controle é enviado apenas um pacote referente a cada rajada. Na Figura 1.9, temos a ilustração de uma rajada de dados associada a um pacote de configuração no tempo. Na ilustração, o **T** indica o tamanho da rajada no tempo, o **δ** indica o intervalo de tempo entre o pacote de configuração e o início da rajada de dados.

Figura 1.9 - Ocupação da rajada no tempo.

A fim de as rajadas serem encaminhadas de forma mais rápida e sem que haja a conversão E/O, o pacote de controle deve ser enviado à frente da rajada com um intervalo de tempo **δ** suficiente entre os dois de forma que o quadro de controle possa ser convertido para o domínio elétrico e processado antes que o *burst* chegue ao nó de comutação, evitando o uso de algum elemento de armazenagem, como buffer ou *Fiber Delay Lines* (FDL).

Existem estudos que avaliam o emprego do método de encaminhamento via rajadas em redes de *datacenter* usando tanto redes OBS, quanto OBS hibrido [23][24].

Dentro do modelo de comutação por rajadas as formas mais conhecidas são, TAG (*Tell and Go*), IBT (*In Band Terminator*), RFD (*Reserve a Fixed Duration*), JIT (*Just In Time*), JET (*Just Enought Time*) e Horizon, que se encontram descritas em maiores detalhes no apêndice A2.

1.6 Conversão Eletro-Óptica

Com a evolução dos sistemas de telecomunicações e necessidade de maiores bandas de dados passou-se a explorar a tecnologia óptica como meio de transmissão de dados devido a várias razões como grande banda de transmissão, imunidade a ruídos eletromagnéticos a baixa atenuação. Um grande destaque do uso da fibra óptica está associado com o desenvolvimento das tecnologias CWDM (Coarse Wavelength Division Multiplexing) e DWDM (Dense Wavelength Division Multiplexing), que possibilitam explorar a enorme largura de banda disponível da fibra óptica, dividindo-a em múltiplos canais.

Embora a tecnologia óptica seja extremamente interessante por vários pontos de vista, muitas vezes não se é possível construir uma rede puramente óptica, sendo necessária a conversão O/E/O (Optical-Eletro-Optical) em algum ponto, uma vez que não existem ainda equipamentos capazes de realizar o processamento da informação diretamente no meio óptico assim como é feito no meio elétrico, para tanto se utilizam de conversores eletro-ópticos.

Como forma de investigar o comportamento de um conversor eletro-óptico foram montados dois experimentos descritos e discutidos nos próximos capítulos.

1.7 Objetivos do trabalho

O presente trabalho tem como objetivo comparação quanto ao desempenho de diferentes formas de manipulação de tráfego, como roteamento IP e encaminhamento de pacotes baseado em Kernel nativo Linux [25][26][27] e empregando Click Modular Router sobre arquitetura de PC. Além de análises comparativas de vazão e latência, o consumo dos recursos de CPU no desempenho das funcionalidades dos equipamentos de comutação foi o principal objeto de pesquisa. Alguns estudos relacionados à vazão e latência são encontrados na literatura em trabalhos como [16], entretanto são estudos que se baseiam no desempenho das redes de *datacenter* como um todo. Como diferencial deste trabalho buscou-se estudar não o desempenho da rede em sua totalidade, mas como um de seus componentes, no caso um servidor, pode ser afetado pelo fluxo de dados que por ele passa e por consequência afetar a rede de *datacenter*. Uma das principais métricas usada aqui foi o uso da CPU dos servidores,

pois tornou possível relacionar o grau de ocupação da CPU com outros parâmetros que medem o desempenho da rede. A forma de encaminhamento, comutação, agregação, pode influenciar na capacidade de vazão de pacotes na rede, como mostram estudos alguns estudos [18]. Tal manipulação também pode refletir na capacidade de processamento dos servidores que podem atuam ora encaminhando, ora processando dados.

1.8 Sumário

Neste capítulo foram citados aspectos relacionados ao atual cenário das redes de computadores e a necessidade de estudos mais atuais referentes ao mesmo. A fim de oferecer uma melhor qualidade de serviço, otimizando o uso da infraestrutura atual e futura, tendo no uso de plataformas de código aberto uma importante ferramenta de desenvolvimento. Também foi apresentado o conceito de redes de datacenter e alguns exemplos de arquiteturas, em particular redes nas quais os servidores passam também a desempenhar atividades de comutação, além de citar e comentar modelos de arquiteturas de transmissão dados, como os de circuitos, pacotes e rajadas, juntamente com a necessidade da conversão entre o meio óptico e elétrico, algo que traz certa limitação ao aproveitamento mais eficaz da banda de dados que o meio óptico disponibiliza.

No próximo capítulo será abordada de forma breve como se dá a manipulação quanto ao envio e recebimento de quadros pelo Kernel do Linux, pois se mostra como estrutura fundamental na manipulação de tráfego por servidores, tomando tal forma de manipulação como referência das outras técnicas investigadas neste trabalho.

Capítulo 2: Linux Kernel Networking

O objetivo do presente capítulo é tratar de forma rápida e simplificada as etapas e tarefas executadas pelo Kernel do Linux [25][26][27] quando o mesmo encontra-se operando com a manipulação de pacotes de dados. Além de apresentar algumas das estruturas de dados usadas no tratamento dos pacotes de dados e também no roteamento ou encaminhamento dos quadros provenientes da rede ou destinados à mesma. Tal estudo busca ajudar a compreender o funcionamento de todo o processo de sinalização e manipulação dos pacotes usando uma estrutura de manipulação básica e facilitar a compreensão das demais técnicas utilizadas mais adiante no trabalho.

2.1 Mapa Kernel Linux

Em se tratando da manipulação de dados de rede pelo Kernel do Linux [25][26][27], pode-se observar por via da Figura 2. 1, onde se encontram as estruturas responsáveis por tais tarefas. Grande parte do código é encontrado em *net/ipv4*. Outra parte relevante é encontrada em *net/core* e *net/sched*. Já os arquivos de header podem ser encontrados em *include/linux* e *include/net*.

O segmento do Kernel se baseia em dois tipos básicos de estruturas, uma para manter a conexão, chamada *sock*, e outra para manter os estados de entrada e saída dos quadros, chamado de *sk_buff*.

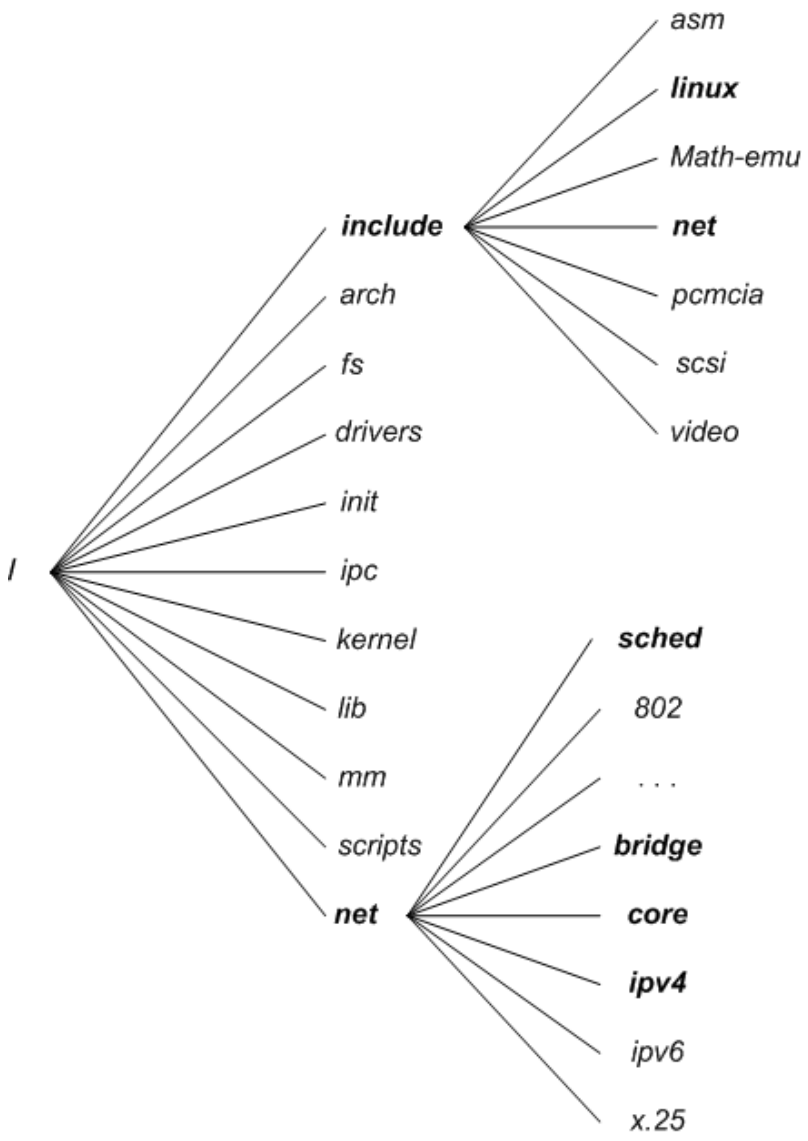

Figura 2. 1 - Mapa Estrutura Linux.

Na Figura 2. 2 encontra-se ilustrada de forma simplificada a estrutura da pilha de rede no Linux. No topo da mesma situa-se a camada conhecida como *User Space*, ou também conhecida como camada de aplicação, funcionando simplesmente de forma a prover a interação entre operações em nível de usuário com as do nível do Kernel. O *User Space* é uma região protegida da memória onde processos de usuário são executados. No nível mais baixo esta o dispositivo físico (NIC – *Network Interface Card*), cuja função é de prover a conexão propriamente dita com a rede, se relacionando diretamente ao protocolo utilizado, como ATM, Ethernet e outros. Entre os dois níveis, encontra-se o local de atuação das

implementações do Kernel (*Kernel Space*). O *Kernel Space* é a região da memória onde os processos do Kernel são executados. A interação entre o *User space* e *Kernel Space* é dada por via do *system call*, transformando um processo do *User Space* em um processo do *Kernel Space*. A pilha de protocolo de rede se baseia quase que totalmente em estruturas responsáveis por toda a manipulação do fluxo de dados. Tais estruturas, os *socket buffers*, são conhecidas por *sk_buff*.

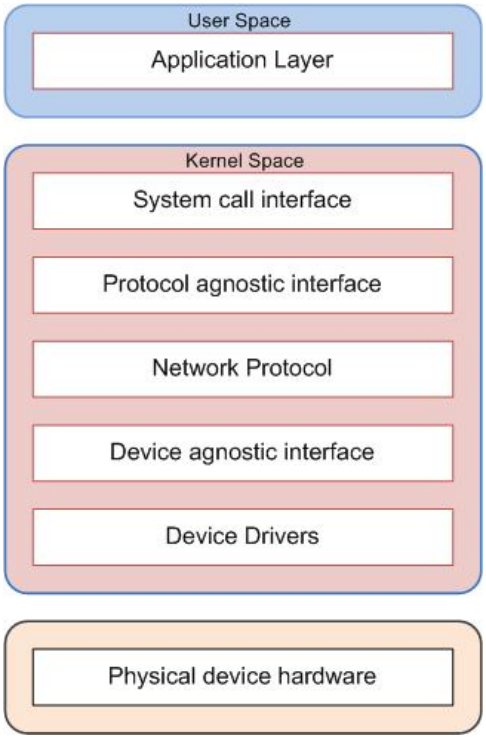

Figura 2. 2 - Pilha da arquitetura de rede do Linux.

A chamada da interface pelo sistema pode ser descrita a partir de duas perspectivas. Quando a chamada de rede é feita pelo usuário, esta é multiplexada através da chamada da interface no Kernel, terminando por chamar o *sys_socketcall* em *./net/socket.c*, o qual demultiplexa a chamada para o seu alvo. A outra perspectiva do sistema se relaciona como a operação de I/O (entrada e saída) da rede. Por exemplo, uma típica operação de escrita e leitura pode ser realizada por um socket, de forma que enquanto o mesmo existe, um conjunto de operações são realizadas, como criação de um socket, como a operação *socket call*, conexão com o destino como *connect call*, etc. Por fim, o *syscall* prove um meio de transferência de controle entre aplicação e o Kernel.

Quanto à camada relacionada ao socket, a mesma provê um conjunto de funcionalidades capaz de dar suporte a diferentes protocolos. Suportando não só protocolos como *Transmission Control Protocol* (TCP) e *User Datagram Protocol* (UDP), mas também *Internet Protocol* (IP), Ethernet e tantos outros como *Stream Control Transmission Protocol* (SCTP). No Linux a estrutura responsável pelo socket é o *struct sock*, sendo o mesmo definido em *linux/include/net/sock.h*. Essa estrutura contém todas as características de um socket, incluindo o protocolo particularmente usado e as operações realizadas. O subsistema responsável pela rede toma conhecimento dos protocolos disponíveis por via de uma estrutura especial que define sua capacidade. Cada protocolo mantém uma estrutura chamada *proto* (*Linux/include/net/sock.h*). Esta estrutura define operações particulares do socket que podem ser realizadas pela camada de socket na camada de transporte, por exemplo, como criar um socket, como estabelecer uma conexão com o mesmo, como encerrar a conexão e terminar o socket.

A seção de protocolo de rede define o protocolo de rede que se encontra disponível ao uso, como TCP e UDP, além de tantos outros, sendo inicializado com a função *inet_init*, *Linux/net/ipv4/af_inet.c*. Esta mesma função *inet_init* atua registrando cada protocolo usado pela função *proto_register*, a qual encontra-se em *linux/net/core/sock.c*. Pode-se analisar os protocolos através da estrutura *proto* em *tcp_ipv4.c*, *udp.c* e *raw.c* encontrados em *linux/net/ipv4*, sendo cada uma das estruturas de protocolos mapeados pelo tipo e protocolo dentro de *inetsw_array*. Na estrutura *inetsw_array*, cada protocolo deste vetor é inicializado com *inetsw* por via da chamada da função *inet_register_protosw* em *inet_init*, a qual também inicializa vários outros módulos de *inet*, como *Address Resolution Protocol* (ARP), *Internet Control Message Protocol* (ICMP), além dos módulos IP, TCP e UDP.

Quando um socket é criado é definido o tipo de socket e o protocolo a ser usado, da seguinte forma *my_sock = socket(AF_INET, SOCK_STREAM, 0)*. O parâmetro *AF_INET* indica uma família de endereço do tipo padrão Internet com o *stream socket* definido como *SOCK_STREAM*. A estrutura *proto* define o método de transporte, enquanto a estrutura *proto_ops* define de forma geral os métodos do socket. Protocolos adicionais podem ser adicionados no protocolo *inetsw* por via da função *inet_register_protosw*. Por exemplo, o SCTP pode ser adicionado através de *scpt_init* em linux/net/sctp/protocol.c.

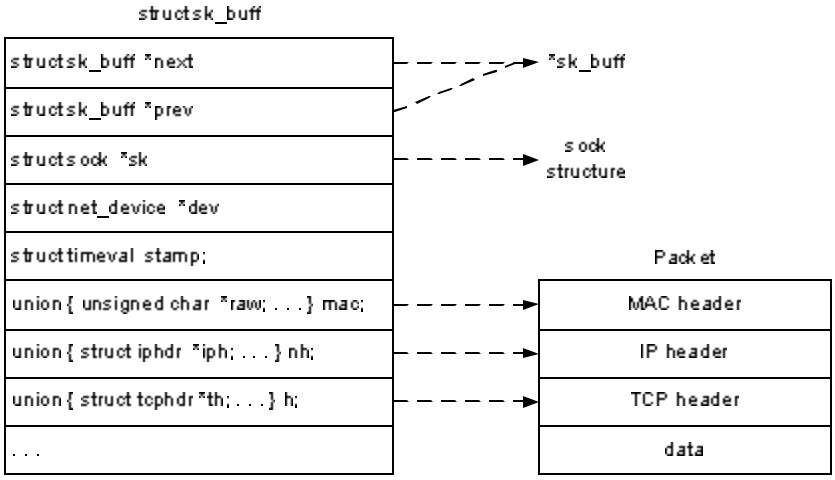

Figura 2. 3 - Estrutura *Socket buffer*.

A manipulação do socket é feita por meio da estrutura de socket buffer *sk_buff*, Figura 2.3. Um *sk_buff* armazena o conteúdo dos quadros, enviados ou recebidos, e o estado das múltiplas camadas da pilha de protocolo. A necessidade de alteração de algum dos campos do quadro recebido pode ser feita por via da alteração dos campos da estrutura *sk_buff*. A estrutura *sk_buff* se encontra definida em *linux/include/Linux/skbuff.h*. Para uma dada conexão, multiplos *sk_buff* podem ser concatenados. Em cada *sk_buff* está identificado pela estrutura *net_device* por onde o quadro foi recebido e por onde será enviado. Como cada quadro é representado por um *sk_buff*, o cabeçalho do pacote se encontra associado a ponteiros, como *th, iph, MAC* e demais campos de cabeçalho. Pode-se perceber que o *sk_buff* constitui como o ponto central da manipulação dos dados recebidos ou enviados pela estrutura de rede, portanto existem funções para criação, destruição, clonagem, manipulação e gerenciamento de fila.

Sob a camada de protocolos existe uma camada, cuja responsabilidade é a de realizar a interface entre os protocolos usados e os *drivers* do hardware, definindo conjuntos a serem utilizados pelos dispositivos de baixo nível de forma a possibilitar que os mesmos possam operar com a pilha de protocolos de rede. O primeiro passo é o registro do dispositivo no Kernel, por via das funções *register_netdevice* ou *unregister_netdevice*, sendo preenchida a estrutura *net_device*, a qual pode ser encontrada em *linux/include/linux/netdevice.h*, e em seguida feito o registro do dispositivo. O Kernel executa a função *init*, executa uma série de checagem, cria um *sysfs*, e em seguida adiciona o novo dispositivo em sua lista de dispositivos ativos *linux/include/linux/netdevice*. Várias funções são implementadas em

linux/net/core/dev.c. Para enviar um *sk_buff* da camada de protocolo para o dispositivo de rede é usada a função *dev_queue_xmit*, função esta que enfileira vários *sk_buff* para um posterior envio ao drive do dispositivo, com o dispositivo de rede sendo definido por *net_device* ou *sk_buff->dev*. A estrutura *dev* contém um método chamado *hard_start_xmit* que atua na transmissão do *sk_buff*. Quando o drive do dispositivo de baixo nível recebe um quadro, o *sk_buff* é repassado para as camadas superiores com a função *netif_rx*, esta função então enfileira os quadros recebidos em *sk_buffs* os quais são repassados a protocolos de nível superior para o posterior processamento com *netif_rx_schedule*. As funções *dev_queue_xmit* e *netif_rx* são encontradas em *linux/net/core/dev.c*. Existe ainda, como alternativa, para realizar a interface com os *drivers* do dispositivo o *New Application Program Interface* (NAPI).

Nas camadas mais inferiores da pilha de rede estão situados os *drivers* que atuam gerenciando os dispositivos físicos de rede. Na inicialização, o dispositivo aloca uma estrutura chamada de *net_device*, que é inicializada com as rotinas necessárias, sendo uma delas a *dev->hard_start_xmit*, definindo como a camada superior deve enfileirar os *sk_buff* para o posterior envio.

A estrutura de dados *sk_buff*, Figura 2. 4, está definida em *include/linux/skbuff.h*. Quando um quadro é processado pelo kernel, uma destas estruturas de dados é criada. A alteração de um campo no quadro é conseguida via alteração no campo da estrutura de dados. Em se tratando do código relacionado com o tratamento da rede, cada função é invocada com um *sk_buff*, comumente chamada por *skb*, passado como parâmetro.

```
struct sk_buff {
        /* These two members must be first. */
    struct sk_buff *next;           /* Next buffer in list */
    struct sk_buff *prev;           /* Previous buffer in list */
    struct sk_buff_head *list;      /* List we are on */
    struct sock *sk;                /* Socket we are owned by */
    struct timeval stamp;           /* Time we arrived */
    struct net_device *dev;         /* Device we arrived on/are leaving by */
```

Figura 2. 4 - Estrutura *sk_buff*.

Na Figura 2.4, os dois primeiros campos, *sk_buff *next* e **prev*, são ponteiros para o *sk_buff* seguinte e anterior respectivamente. Os pacotes recebidos ou a serem transmitidos encontram-se armazenados em filas e as estruturas ilustradas na Figura 2.4, como *sk_buff_head*, *sock*, *time stamp* e *net_device*, possibilitam a sua identificação.

Outros tipos de informações referentes ao pacote de dados são exemplificadas pela Figura 2. 5.

```
char cb[48];
unsigned int len;                /* Length of actual data */
unsigned int data_len;
unsigned int csum;               /* Checksum */
unsigned char __unused,          /* Dead field, may be reused */
cloned,                          /* head may be cloned (check refcnt to be sure) */
pkt_type,                        /* Packet class */
ip_summed;                       /* Driver fed us an IP checksum */
__u32 priority;                  /* Packet queueing priority */
atomic_tusers;                   /* User count - see datagram.c,tcp.c */
unsigned short protocol;         /* Packet protocol from driver */
unsigned short security;         /* Security level of packet */
unsigned int truesize;           /* Buffer size */
unsigned char *head;             /* Head of buffer */
unsigned char *data;             /* Data head pointer*/
unsigned char *tail;             /* Tail pointer */
unsigned char *end;              /* End pointer */
```

Figura 2. 5 - Campos de informação sobre os Pacotes de dados.

Em se tratando do gerenciamento de memória, a atribuição do descritor do pacote é feita em *net/core/skbuff.c* pela função *alloc_skb( )*, usada cada vez que um novo *buffer* é necessário. A função recebe o cabeçalho do conjunto de pacotes do processo atual (*skb_head_from_pool*), atribui a memória para a carga de dados com *kmalloc( )* e ajusta-se o ponteiro de dados e o estado do descritor, além de recolher algumas estatísticas de memória para depurar aspectos de memória.

Alguns pacotes são atribuídos através *skb_clone( )* quando apenas os meta-dados (no *sk_buff struct*) precisam ser duplicados para os pacotes de dados. Este é o caso, para pacotes TCP/IP no lado transmissor. A diferença entre os dois tipos de alocação está na desalocação, SKB é alocada por *alloc_skb(),* desalocados no momento da chegada do ACK, enquanto os atribuídos pelas *skb_clone( )* são desalocados depois de receber e transmitir eventos de conclusão do NIC (Network Interface Card). A desalocação de *sk_buff* é feita pela função *__kfree_skb( )*. Ela libera o campo DST com *dst_release( )*. Este campo contém, entre outras coisas, o dispositivo de destino do pacote. A função chama *skb->destructor( )*, se presente, para fazer algumas operações específicas antes da limpeza. Desalocar um *skb* finalmente envolve a limpeza (para reutilização futura) com *skb_headerinit( )*, liberando a parte de dados, se não for um clone, e inserindo-o em um *skb* livre para reutilização futura com *kfree_skbmem*.

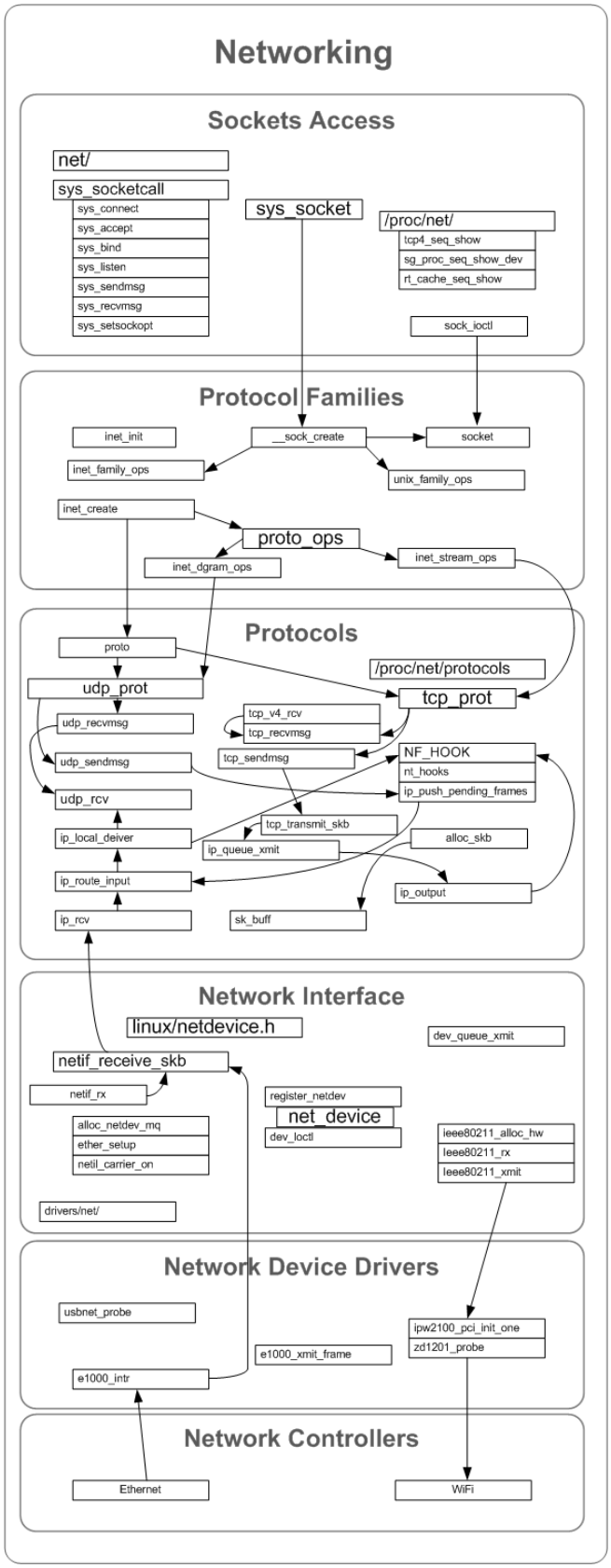

Figura 2. 6 - Mapa das estruturas de rede do Kernel.

Algo importante é a existência dentro do conjunto de códigos e estruturas responsáveis pelas atividades de rede no kernel de vários ganchos, *netfilter*, onde desenvolvedores podem associar seu próprio código e analisar ou alterar pacotes. A Figura 2. 6 apresenta uma visão geral do que o pacote passa através do Kernel. Ela indica as áreas onde o hardware e o código do driver operam, o papel da pilha de protocolo do kernel e da interface kernel /aplicação.

2.2 O NIC no processo de envio e recepção de quadros

No processo de recepção e envio de quadros de e para a rede estão envolvidos além de todas as estruturas e funções do Kernel já ilustradas e comentadas anteriormente, também etapas que se relacionam diretamente com a interface de rede presente no próprio computador, como manipulação de memória, controle de barramento, sinalização e acesso direto a memória. As Figuras 2.7 e 2.8 ilustram de forma resumida as etapas de envio e recepção de um quadro.

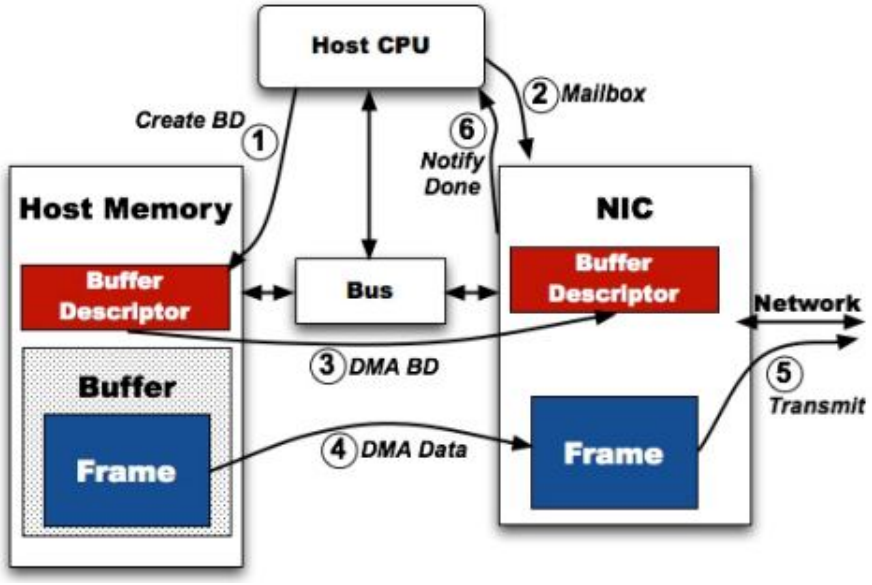

Figura 2. 7 - Etapas para o envio de um único quadro [28].

1. É informado ao sistema operacional que o quadro está na memória e já está pronto para ser enviado. O Sistema Operacional (OS) monta um buffer de acordo com o quadro na memória a ser enviado;

2. O NIC recebe uma notificação do SO de que há um buffer na memória pronto a ser buscado e processado;
3. A interface NIC inicia a leitura e processo do buffer via Acesso Direto à Memória (DMA);
4. Tendo já determinado o endereçamento do quadro, inicia-se por DMA a leitura do conteúdo do quadro;
5. Quando é terminada a leitura de todos os segmentos que compõem o quadro, o mesmo é enviado pelo NIC;
6. Dependendo da configuração pode-se gerar uma interrupção a fim de indicar o completo envio do quadro;

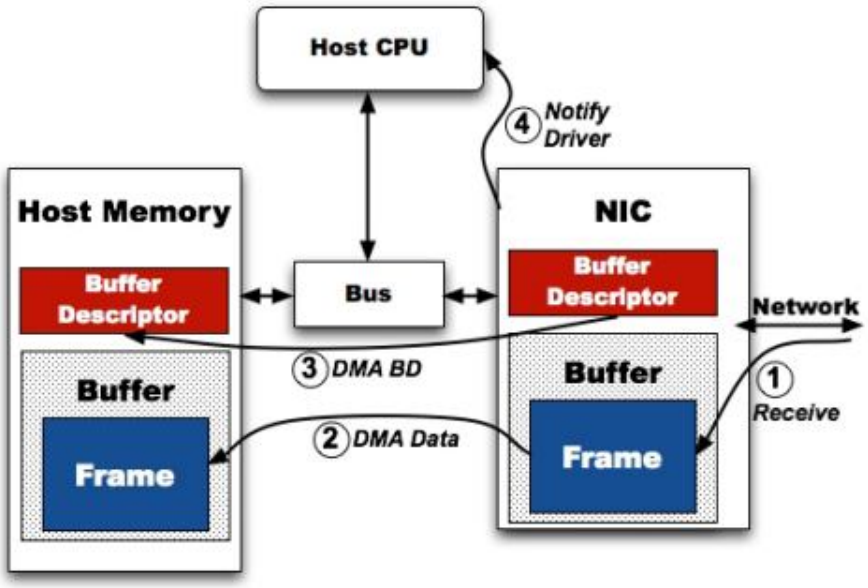

Figura 2. 8 - Etapas de recepção de um quadro [28].

1. O NIC recebe o quadro vindo da rede e o armazena em um buffer;
2. O NIC começa a escrever o conteúdo do quadro via DMA na memória;
3. O NIC modifica o buffer de acordo com o quadro recebido, determinando onde se inicia o cabeçalho e preenche o *checksum* do quadro;
4. Pode ser gerada uma interrupção para informar que o quadro foi totalmente recebido.

Tendo feita toda a análise do processo de manipulação dos pacotes pelo Kernel do sistema operacional e também da descrição básica das etapas realizadas pela interface de rede no recebimento e envio de pacotes. As Figuras 2.9 e 2.10 podem ilustrar, de forma resumida, o processo de recepção, manipulação e entrega dos dados à aplicação final, Figura 2. 9. Também é possível observar o caminho inverso dos dados desde a aplicação, passando pela manipulação do Kernel, até o envio pela interface de rede, Figura 2. 10. As etapas ilustradas de forma resumida apresentam uma visão global dos procedimentos realizados pelo Kernel durante o procedimento de recepção, manipulação e envio do pacote de dados.

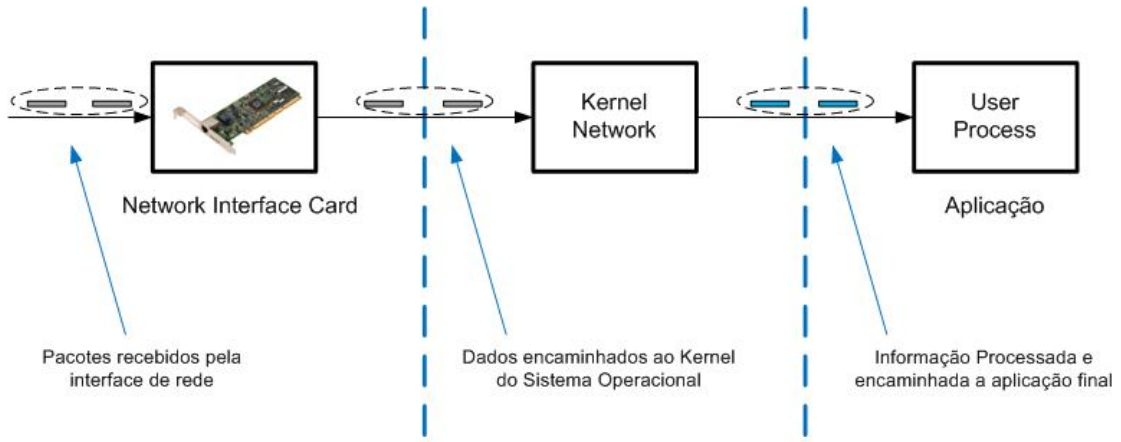

Figura 2. 9 - Manipulação pelo Kernel dos pacotes de rede recebidos.

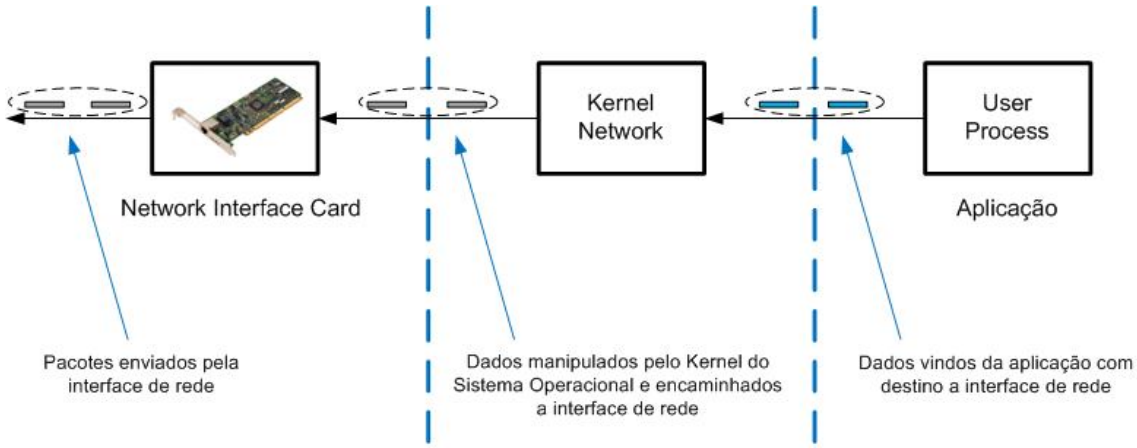

Figura 2. 10 - Manipulação pelo Kernel dos pacotes de rede enviados.

Nas Figuras 2.9 e 2.10 observa-se que todos os pacotes com origem ou destino a rede são em algum momento manipulados pelo Kernel do sistema operacional.

2.3 Sumário

Neste capítulo foi descrita de forma simplifica o modo como o Kernel do Linux trata os quadros recebidos a partir da interface de rede, além de como a própria interface atua na recepção e envio dos quadros Ethernet. Também foram exemplificadas algumas das estruturas mais importantes usadas em tais tarefas de manipulação. Outra questão tratada de forma breve é referente às etapas realizadas tanto no envio quanto na recepção dos quadros destinados ou provenientes da rede.

Capítulo 3: API Click e o uso de elementos

A arquitetura *Click* é baseada em *elementos*. Cada elemento é um componente de software que representa uma unidade de processamento. Elementos realizam operações conceitualmente simples, tais como um decremento do campo *Time-to-live* no pacote IP, em vez de grandes cálculos complexos, como o próprio roteamento. Eles geralmente examinam ou modificam pacotes de alguma forma. Pacotes são estruturas de dados de rede que os equipamentos de rede processam. Em tempo de execução, os *elementos* passam pacotes para uma outra através de links chamados conexões. Cada conexão representa um caminho possível para a transferência de pacotes [8].

3.1 Click Modular Router

A proposta da arquitetura CLICK é de ser uma ferramenta flexível, modular e de alto desempenho em se tratando do encaminhamento e análise de pacotes no âmbito de redes de computadores. A ideia por trás do Click se liga à manipulação via software do hardware de um simples computador, dando ao mesmo as funcionalidades de equipamentos de rede de computadores, o que leva ao fato de diversos aspectos desta arquitetura terem sido inspirados diretamente por propriedades de roteadores [9]. Esta relação pode ser observada já no modo como repassa os pacotes ao longo de uma conexão, podendo ser iniciada tanto pela extremidade de origem (pacote é 'empurrado') quanto pela extremidade de destino (pacote é 'puxado'), modelando a maior parte dos padrões de fluxo de pacotes em roteadores. O software CLICK se apresenta em duas formas distintas dependendo da aplicação, podendo ser como aplicação em nível de usuário (driver *user-level*), ou como módulo do em nível kernel Linux (*driver linux-module*). As duas possibilidades se destinam a funções diferentes. O *driver user-level* é útil para depuração e execução de testes repetitivos, ele pode receber pacotes da rede usando mecanismos do sistema operacional através de bibliotecas originalmente criadas para *sniffers*. Entretanto, o *user-level* se restringe a pilha de rede do

sistema operacional para encaminhar um pacote, apresentando limitações quanto ao desempenho uma vez que se encontra submisso ao sistema operacional (SO). Já o *driver* em nível de Kernel, o *linux-module*, substitui completamente a pilha de rede do SO, transformando um computador pessoal convencional num roteador ou outro elemento de rede, alcançando alto desempenho [9]. Portanto, devido à tal capacidade fez-se opção pelo uso da última forma de *driver* neste trabalho, o *linux-module*.

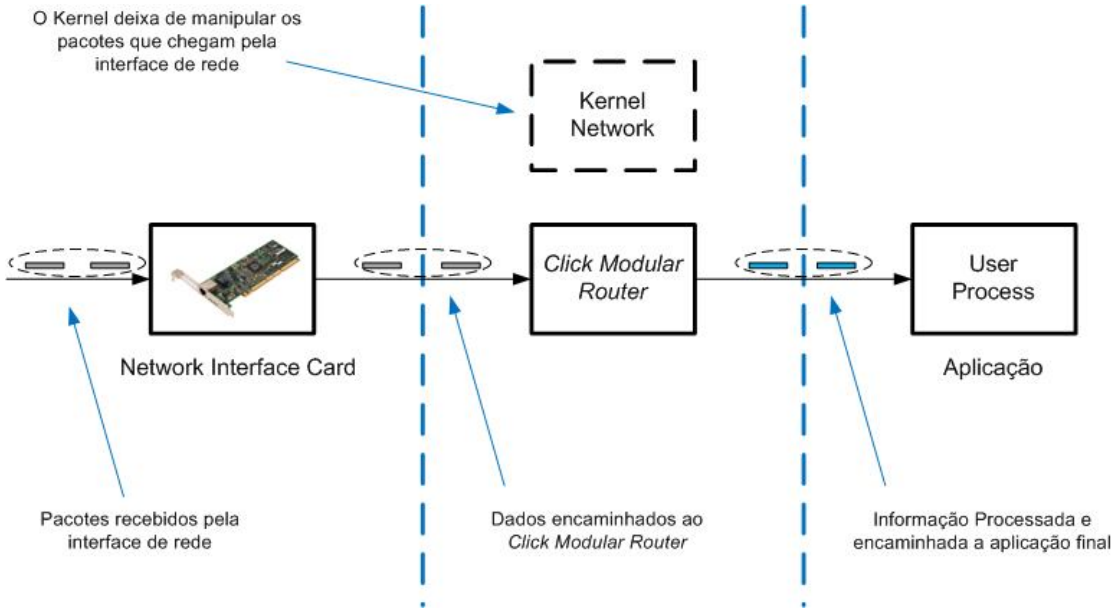

Figura 3. 1 - Manipulação pelo Click Modular Router dos pacotes de rede recebidos pela interface de rede.

Como se pode observar a partir da ilustração das Figuras 3.1 e 3.2, o Click Modular Router executado com o *drive kernel-module* se sobrepõe aos mecanismos da pilha TCP/IP do Kernel do próprio sistema operacional (SO), capturando os pacotes recebidos diretamente da interface de rede (Network Interface Card – NIC) e direcionando os mesmos para a interface de saída apropriada após a análise dos mesmos sem também nenhuma participação das bibliotecas específicas do *Kernel* do SO.

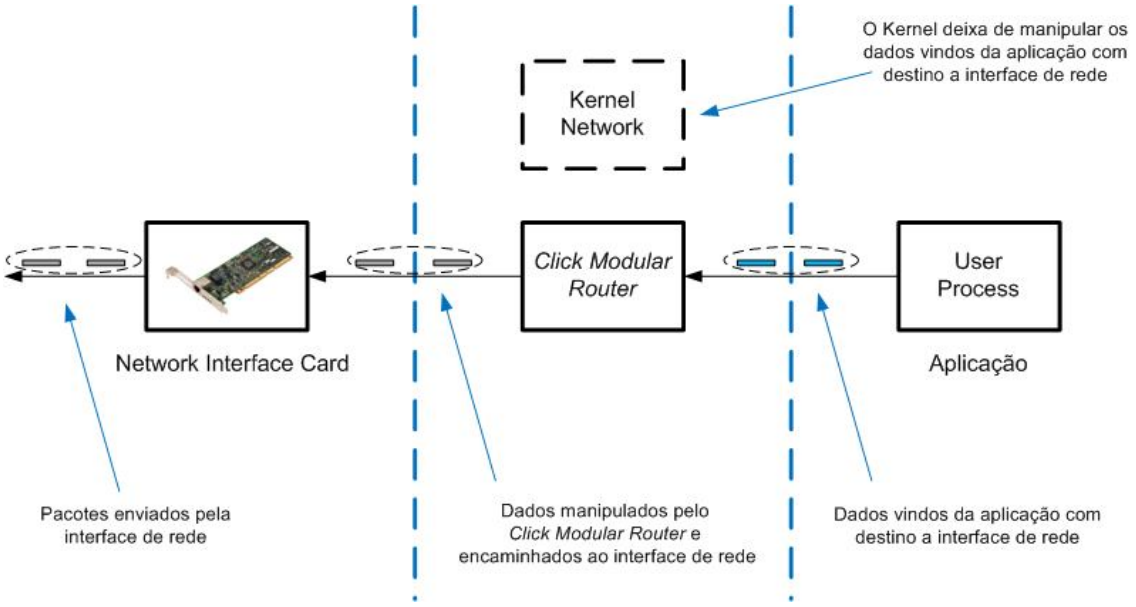

Figura 3. 2 - Manipulação pelo Click Modular Router dos pacotes de rede enviados pela interface de rede.

3.1.1 Click Kernel module

O Click Modular Router pode ser compilado e carregado pelo sistema operacional como um módulo do Kernel, ***click.o***. No modo Kernel, o *Click Modular Router* adquire total autonomia sobre a manipulação dos pacotes que entram ou saem das interfaces de rede, antes mesmo que o próprio Linux tenha a oportunidade de manipular os mesmos. A instalação de um arquivo de configuração Click em modulo Kernel se dá via o seguinte comando: *click-install arquivo.click*, podendo ser associados a alguns parâmetros, já para desinstalar uma configuração que esteja sendo executada no modulo Kernel, basta executar: *click-uninstall*.

Caso o usuário tenha um arquivo de configuração do Click sendo executado em modulo Kernel e queira executar um novo arquivo com algumas alterações, não é necessário aplicar um *click-uninstall* seguido de um *click-install novo_arquivo.click*, basta apenas executar *click-install –h novo_arquivo.click*, uma vez que o parâmetro "*-h*" faz referência a ferramenta *hotswap*. O *hotswap*, durante a execução do arquivo, carrega apenas os elementos que foram alterados e caso a alteração acarrete algum erro de configuração o arquivo anterior permanece em funcionamento, já na ausência do parâmetro "*-h*" caso o novo arquivo possua

algum erro de configuração o mesmo não poderá ser instalado e o anterior em execução não mais permanecerá em execução.

3.2 Elementos *Click Modular Router*

Como já mencionado anteriormente, a base de trabalho com *Click Modular Router* está fundamentada na manipulação e associação de blocos funcionais denominados *elementos*. Os elementos são na verdade estruturas de objetos criados em linguagem C++, usando biblioteca padrão e específica do *Click*. Como estrutura básica de um elemento tem-se a definição das portas de entrada e saída e o tipo da mesma, podendo ser tipo *push*, *pull* ou *agnostic*. Quando da associação dos elementos, os mesmos só podem ser ligados segundo a seguinte regra: semelhante se conecta à semelhante, ou seja, um elemento *A* com porta de saída *push* só pode ser ligado a outro elemento *B* cuja porta de entrada também seja do tipo *push*, a exceção se faz presente para o caso de um elemento *agnostic*, o qual por característica pode ser conectado tanto a elementos *push*, quanto *pull*, possibilitando assim a interligação de diferentes tipos de elementos. Na Figura 3.3 tem-se um simples exemplo de conexão entre elementos no *Click*. O elemento *FromDevice*, que captura pacotes que chegam a interface *eth0*, possui um porta se saída do tipo *push*, portanto tal elemento só poderá se ligar à outro elemento que possua porta de entrada do mesmo tipo *push*, no caso o elemento *Null*. Já o elemento *Queue*, por possuir portas do tipo *agnostic*, pode se ligar tanto a elementos com portas tipo *push* quanto *pull*.

Figura 3. 3 - Conexão entre elementos.

Para a construção de um roteador utilizando a tecnologia CLICK é necessária uma coletânea de elementos, que serão interconectados através de uma linguagem visual de configuração em blocos de elementos.

Assim, para expandir a arquitetura para dar suporte a novas funções de roteamento, tais como novos protocolos, basta modificar os elementos já existentes ou criar novos elementos.

As propriedades mais importantes de um elemento são:

- *Classe do elemento*: cada elemento pertence a uma classe específica que indicará o código a ser executado no processamento de um pacote, assim como a função de iniciação do elemento e seu formato de dados.
- *Portas*: Um elemento pode ter portas de entrada e saída em qualquer número, inclusive nenhuma. Toda conexão, entre os elementos, vai de uma porta de saída de um elemento até a porta de entrada em outro. Para que se possa iniciar uma configuração todas as portas dos elementos devem estar conectadas.
- *String de configuração*: é opcional e pode conter argumentos de configuração que são iniciados junto com a configuração do roteador. Muitas classes de elementos utilizam estes argumentos para realizar um ajuste fino no comportamento de um elemento.
- *Interface*: Cada elemento dá suporte a uma ou mais interfaces. Todo elemento dá suporte a interface de transferência de pacotes, mas os elementos podem criar e exportar interfaces adicionais arbitrárias; por exemplo, uma fila pode exportar uma interface que relate seu comprimento.
- *Handler*: É possível ao usuário verificar parâmetros e contadores do roteador instalado, similarmente aos roteadores comerciais, através destes *handlers*. Quando uma configuração de roteador é instalada esses parâmetros são acessíveis em arquivos ASCII em um diretório criado dinamicamente com todas as instâncias dos elementos descritas na configuração. Assim, basta realizar um *cat* no arquivo do *handler* desejado, como por exemplo, para saber a ocupação de uma fila basta utilizar o comando *cat /click/Queue/Length* no terminal da máquina onde é instalado o roteador.

3.3 Exemplos de uso do *Click Modular Router*

Uma forma de conhecer de forma mais realista a arquitetura *Click Modular Router* é com a implementação e análise de cada elemento individualmente e de todo o conjunto que forma o arquivo de configuração. Portanto, de forma a dar um melhor esclarecimento sobre o

mesmo, na sequência serão descritos e ilustrados exemplos de como se dá a construção no *Click Modular Router* de um roteador IP e de um comutador.

3.3.1 Exemplo 1: Construção de um Comutador

A fim de garantir o encaminhamento do pacote, o primeiro passo é a sua captura diretamente na interface da rede, evitando que o mesmo seja tratado pelo sistema operacional, através do elemento *FromDevice*. Na sequencia o quadro é classificado quanto ao seu tipo e posteriormente encaminhado para a interface de destino por meio do elemento *ToDevice*, conforme ilustrado na Figura 3.4.

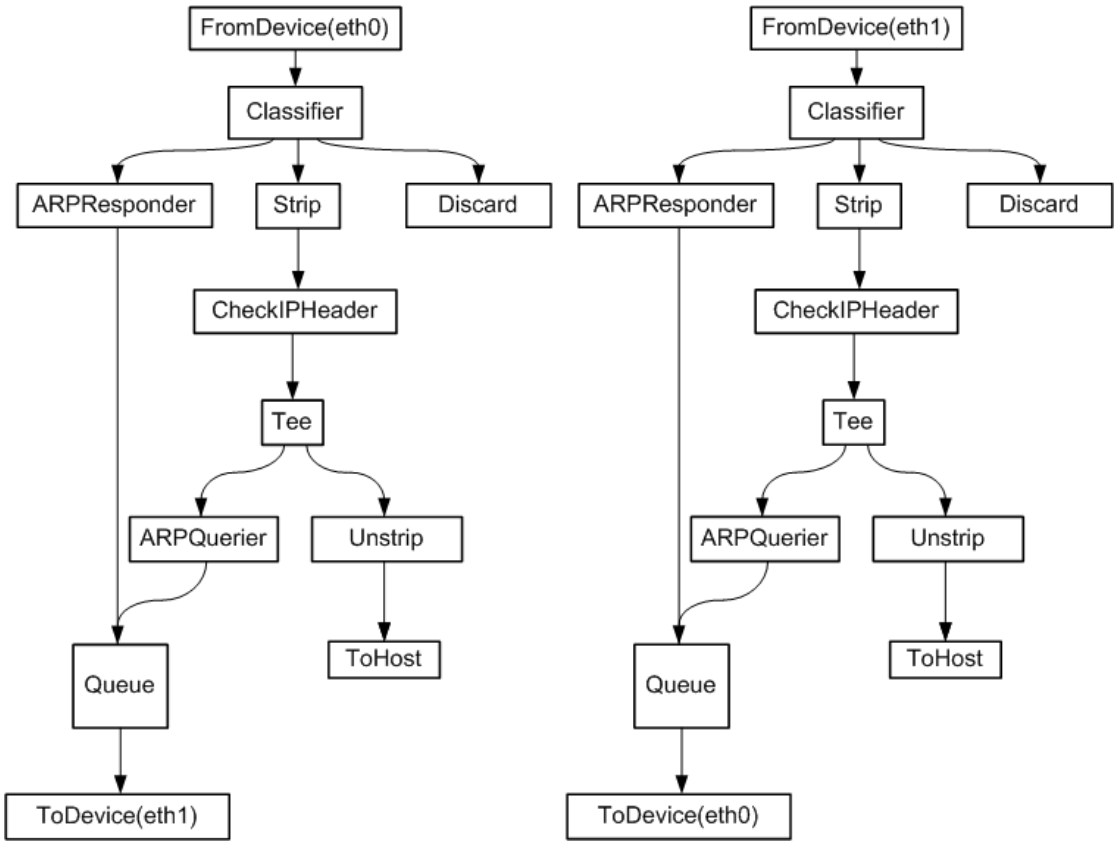

Figura 3. 4 - Comutador com duas entradas e duas saídas.

Quanto aos elementos utilizados tem-se:

- *Classifier*: possui uma porta de entrada e N de saídas, cada uma associada ao um padrão de configuração, de forma a separar, classificar, os quadros com base nas informações de seu cabeçalho. Na Figura 3. 5 tem-se um exemplo do modo de configuração do elemento *Classifier( )*:

```
c0 :: Classifier(12/0806  20/0001,        // Identifica Pedidos ARP
                 12/0806 20/0002,         // Identifica Respostas ARP
                 12/0800,                 //Identifica pacote do tipo IP
                 -);                      //Qualquer outro tipo de pacote
```

Figura 3. 5 - Configuração do *element Classifier*.

- *ARPResponder*: responde as requisições das mensagens ARP. Na Figura 3. 6 tem-se um exemplo básico de configuração do elemento *ARPResponder( ),* com o endereço IP, mascara de rede e endereço MAC (*Ethernet*) da interface de rede:

```
ar0 :: ARPResponder(end. IP/Mascara end. Ethernet);
```

Figura 3. 6 - Configuração do *element ARPResponder*.

- *ARPQuerier*: encapsula os quadros IP com o cabeçalho Ethernet, preenchendo o campos com os endereços de origem e destino. Na Figura 3. 7 tem-se um exemplo de configuração do elemento *ARPQuerier( ),* onde se utilizam dos endereços IP e MAC (*Ethernet*):

```
arpq0 :: ARPQuerier(end. IP, end. Ethernet);
```

Figura 3. 7 - Configuração do *element ARPQuerier*.

- *Tee*: age duplicando os pacotes que entram em sua interface. Possui uma porta de entrada e *N* portas de saídas, para cada interface de saída é enviado uma copia do pacote. Na Figura 3. 8 tem-se um exemplo simples de configuração do elemento *Tee( )*:

```
t :: Tee( N );
```

Figura 3. 8 - Configuração do *element Tee*.

3.3.2 Exemplo 2: Construção de um simples Roteador IP

Como primeiro passo na manipulação de pacotes, é necessário capturá-los e na sequência identificar o conjunto de informações contidas em seu cabeçalho. Um elemento responsável por essa tarefa é o *FromDevice( )*, que atua capturando o quadro diretamente da interface de rede especificada. Na sequencia o quadro é classificado segundo o tipo de quadro pelo elemento *Classifier* em pacotes *ARP Queriers*, *ARPResponses* e pacotes IP, tendo cada tipo de quadro o seu devido tratamento. Os pacotes *ARPQueries* são encaminhados ao *ARPResponder*, cuja tarefa é de responder as requisições ARP pela interface por onde o quadro foi capturado. Já os pacotes de resposta ARP são encaminhados ao Linux, e também ao elemento *ARPQueriers* do roteador, salvando estas informações nas tabelas ARPs no caso de a requisição ter sido realizada pelos mesmos. Por fim, os pacotes IP são marcados e encaminhados para o roteamento. A implementação esquemática de um roteador IP no *Click Modular Router* encontra-se ilustrada na Figura 3.9.

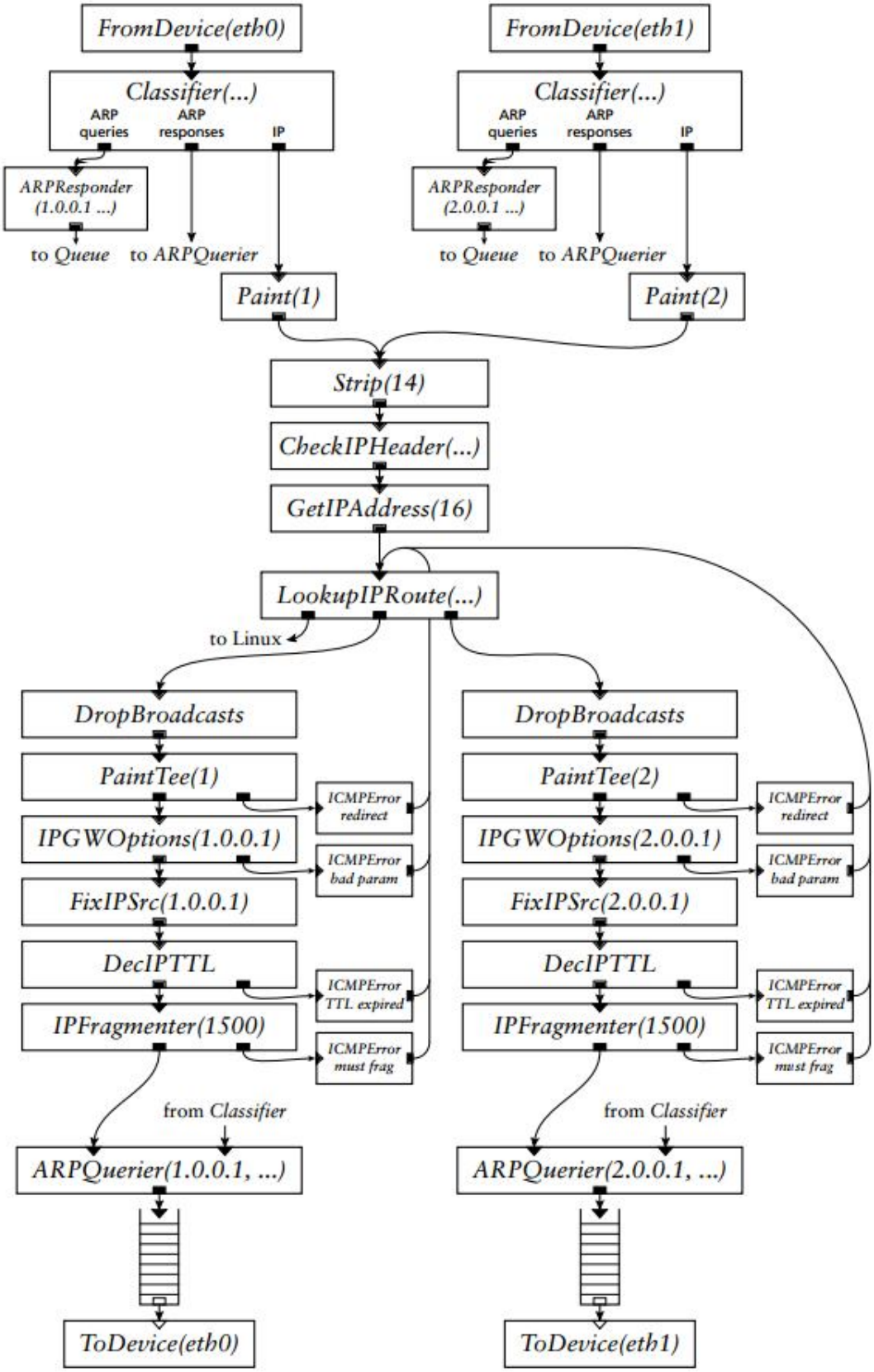

Figura 3. 9 - Simples implementação em CLICK de Roteador IP [9].

Tarefas de roteamento por IP que envolvam somente informações locais se encaixam naturalmente na estrutura CLICK. Por exemplo, *DecIPTTL* verifica se o campo TTL de um pacote expirou. Se o TTL ainda for válido, *DecIPTTL* o decrementa, atualiza o *checksum* do pacote e envia o mesmo por sua primeira saída; se o TTL expirou, *DecIPTTL* envia o pacote por sua segunda saída (conectada geralmente a um elemento que gere erros ICMP). Essas ações dependem somente do conteúdo dos pacotes, e não interagem com as decisões tomadas em outras partes. Elementos como o *DecIPTTL* podem ser facilmente compostos. Por exemplo, poderíamos conectar a saída "expirada" de *DecIPTTL* a um *Discard* para evitar a geração de erros ICMP. Entretanto, algumas tarefas de transferência requerem que informação sobre um pacote seja calculada em um lugar e usada em outro. CLICK usa um artifício denominado anotações para carregar tais informações. Uma anotação é um pedaço de informação colocada no cabeçalho do pacote, mas não faz parte dos dados do pacote; a própria estrutura de pacotes do Linux, *sk_buff*, fornece 48 B de espaço para anotações. As anotações usadas no roteador IP incluem:

- *Destination IP address*: Elementos que lidam com o endereço IP de destino de um pacote usam esta anotação ao invés do campo de cabeçalho do IP. Isso permite que os elementos modifiquem o endereço de destino para ajustar-lo ao endereço do gateway do próximo salto, por exemplo sem modificar o pacote. O *GetIPAddress( )* copia o endereço do cabeçalho IP para a anotação, *StaticIPLookup( )* substitui a anotação com o endereço do gateway do próximo salto, e *ARPQuerier( )* mapeia a anotação para o endereço Ethernet do próximo salto.

- *Paint*: O elemento do tipo Paint marca cada pacote com um inteiro "cor". *CheckPaint* emite todo pacote em sua primeira saída, e uma cópia de cada pacote com uma cor específica em sua segunda saída. O roteador por IP usa essa “cor” para saber se um pacote está saindo pela mesma interface de rede em que chegou, e assim saber se deve gerar um erro de ICMP *redirect*.

- *Link-level broadcast flag*: *FromDevice* configura essa flag nos pacotes que chegaram como broadcast em nível de link. O roteador por IP usa *DropBroadcast* para descartar tais pacotes se estiverem a ponto de serem enviados a uma outra interface.

- *FixIPSource flag*: O endereço IP da fonte de um pacote de erro ICMP deve ser o endereço da interface em que o erro é emitido. ICMPError não pode prever esta interface, assim ele usa um endereço padrão e ajusta a anotação FixIPSource flag. Depois que o pacote ICMP for roteado para uma interface particular, um FixIPSrc nesse trajeto verá a *flag*, introduzirá o endereço IP correto da fonte, e recalculará o *checksum* do pacote.

- *StaticIPLookup:* Implementa uma simples tabela de roteamento IP, a qual é preenchida com os endereços de para o roteamento juntamente com suas mascaras de rede, seguidos pela identificação de por qual porta do elemento que certo pacote irá ser encaminhado. Na Figura 3. 10 tem-se um exemplo de configuração do elemento *StaticIPLookup( )*:

```
rt :: StaticIPLookup(end. IP 1/Mascara porta de saída 0,
                     end. IP 2/Mascara porta de saída 1,
                     end. IP 3/Mascara porta de saída 2,
                     end. IP 4/Mascara porta de saída 3,
                     end. IP 5/mascara porta de saída 4);
```

Figura 3. 10 - Configuração do *element StaticIPLookup*.

Este exemplo de roteador baseado em *Internet Protocol* (IP) mostrado conforma-se aos padrões do roteamento tanto por causa do comportamento individual dos elementos quanto por seu arranjo. Em termos de elementos individuais, o elemento *CheckIPHeader* verifica exatamente as propriedades exigidas pelos padrões: os campos de comprimento do IP, o endereço IP da fonte, e o *checksum* do pacote. *IPFragmenter* fragmenta os pacotes maiores do que o MTU (Maximum Transmission Unit) configurado, mas emite pacotes grandes, não fragmentados, a uma saída de erro. Finalmente, *ICMPError*, que encapsula a maioria dos pacotes de entrada em uma mensagem de erro ICMP e coloca o resultado em sua saída, não responde a *broadcasts*, erros ICMP e fragmentos; não devem ser gerados erros ICMP em resposta a esses pacotes. Em termos das propriedades de arranjo dos elementos, os padrões exigem a colocação de *DecIPTTL* após o elemento de roteamento *StaticIPLookup*. O TTL de um pacote só pode ser decrementado depois que se determina que o pacote não está destinado para o próprio roteador.

3.4 Sumário

Neste capítulo foram tratados de assuntos relativos à API *Click Modular Router*. Em um primeiro momento foi comentado o que constitui a API, seus modos de execução, seus elementos fundamentais e posteriormente foram exemplificados e comentados dois exemplos de configuração do Click, um simples comutador e um roteador IP. A ferramenta Click Modular Router se mostrou interessante a este trabalho devido a sua flexibilidade na construção de estruturas capazes de manipular pacotes de rede a partir de seus blocos funcionais mais simples.

Dando prosseguimento ao trabalho, no próximo capítulo serão tratados os métodos usados para a coleta de dados nas implementações em laboratório.

Capítulo 4: Metodologia e Testes

No decorrer deste trabalho foram citados alguns métodos de conformação de tráfego de dados, citando suas características e funcionamento, de forma a propiciar uma discussão relativa ao seu melhor emprego. Foi realizado também um estudo sobre a ferramenta *Click Modular Router,* de forma a justificar a sua aplicabilidade e viabilidade em se tratando do uso nos experimentos em laboratório. Na sequência, serão descritos e ilustrados de forma detalhada todos os procedimentos utilizados no decorrer do trabalho. Procedimentos estes que tornaram possível a discussão sobre como a forma de manipulação dos pacotes na rede pode impactar sobre demais atividades dos servidores e sobre a própria rede.

4.1 Objetivos

O objetivo desta dissertação é realizar uma análise da forma como a manipulação do fluxo de dados pode comprometer os recursos de uma rede de *datacenter* centrada em servidores. Particularmente a análise é feita nos casos em que os servidores além de realizar atividades de processamento de dados, também passam a atuar como encaminhadores. Para tanto, houve a necessidade de se construir uma infraestrutura de experimentos, *setup* de testes, fisicamente utilizando equipamentos disponíveis em laboratório, de forma a criar um ambiente onde fosse possível a obtenção de dados de forma mais realista possível, com todas as limitações impostas por um ambiente real, algo que em geral não se obtém em um ambiente simulado, pois no mesmo as limitações são de antemão estabelecidas no momento da modelagem do ambiente.

4.2 Bancada de experimentos

De forma a realizar os experimentos, foi montado o *setup* ilustrado pela Figura 4.1, composto por cinco computadores, sendo quatro possuindo cada um quatro interfaces ethernet. Em cada um dos quatro computadores, denominados *Click-1*,*Click-2*, *Click-3 e Click-4*, foi utilizada a distribuição Linux Debian, com Kernel versão 2.6, pois a mesma é sugerida pelo próprio desenvolvedor do *Click Modular Router*. Os quatro são computadores antigos que já não se encontravam em uso, todos com processadores Intel Pentium IV, 80 Gbytes e 512 ou 1024 Mbytes de memória. De forma a possibilitar a flexibilidade, foi instalado o *Click Modular Router* no modo Kernel, *kernel-module*, em cada um dos computadores, realizando as adaptações necessárias, como compilação do Kernel, instalação de pacotes extras e aplicação do *patch* fornecido pelo próprio desenvolvedor. No quinto computador, denominado *Servidor*, foram instaladas máquinas virtuais com a distribuição Ubuntu 11.0 e Kernel de versão 2.30, gerenciadas pelo software de distribuição gratuita *VirtualBox* e configurado de forma que cada uma das máquinas virtuais fosse atrelada a cada uma das interfaces Ethernet do servidor, de forma a possibilitar a criação de fluxo de dados distintos, gerados em interfaces distintas, mas regidos pelo mesmo relógio, já que fisicamente trata-se de uma única máquina.

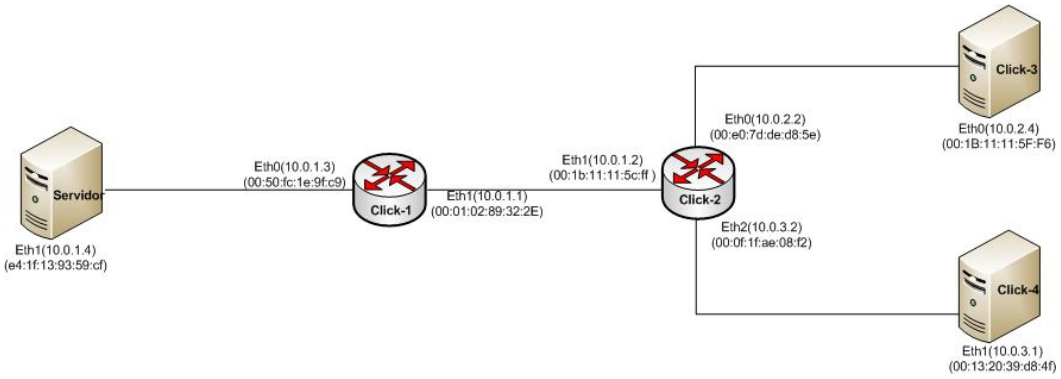

Figura 4.1 - Setup Montado em laboratório.

As montagens foram feitas de forma a usar as máquinas, computadores, como fonte, destino e encaminhadores ou roteadores de tráfego.

Nas Figura 4.1 e 4.2, encontra-se ilustrada uma montagem com todas as cinco máquinas, sendo uma, o *Servidor*, usada como gerador ou fonte, as intermediárias, *Click-1* e

2, desempenhando atividade de encaminhadores, e por ultimo as máquinas *Click-3* e *4* como destinos do trafego de dados.

O setup ilustrado pela Figura 4.1 foi escolhido, pois o mesmo com cinco máquinas apenas possibilitam duas máquinas de destino diferentes, *Click-3* e *4*, além de pontos na rede, *Click-1* e *2*, onde seria possível a aferição do consumo de recursos e desempenho tornando, além da possibilidade de comparação dos resultados apresentados por cada uma das máquinas.

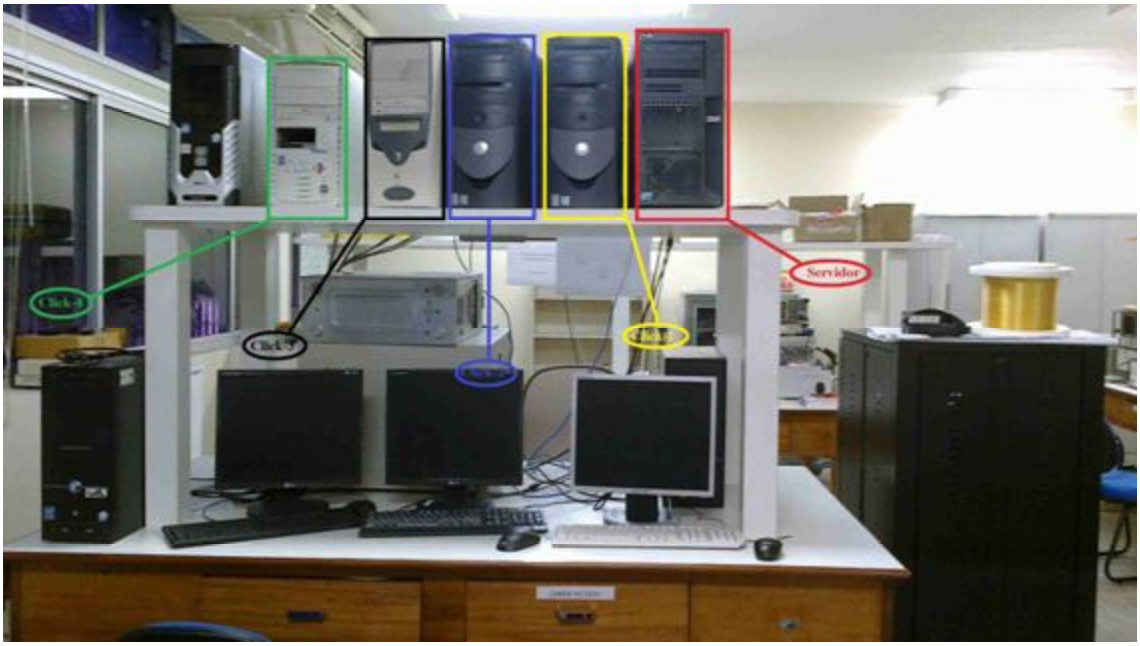

Figura 4. 2 - Setup montado em laboratório.

4.3 Metodologia

Como metodologia de experimentos foi montado em laboratório o *setup* já descrito e utilizado software livre para a criação e gerenciamento de fluxos tanto em *tcp* quanto em *udp*, sendo testados três métodos de encaminhamento, descritos a seguir.

4.3.1 Roteamento IP, *Linux Kernel*

A fim de montar o *setup* para teste do roteador IP implementado diretamente no Kernel do sistema operacional foram criadas rotinas de encaminhamento com o *IPtable* [29], de forma a realizar o encaminhamento baseado no endereço IP de destino, assim como um roteador. Nos testes realizados foi utilizado o mesmo *setup* ilustrado na Figura 4.1, onde nas máquinas Click-1 e Click-2 foram executadas rotinas de encaminhamento baseadas em *IPtable*.

4.3.2 Roteamento IP, *Click Modular Router*

Com o objetivo de avaliar o desempenho do roteamento baseado no protocolo IP, contornar e identificar alguma possível limitação própria do *Click Modular Router* foi construído com o mesmo um roteador IP. O roteador foi montado aplicando se pequenas adaptações, passando a usar três interfaces de rede ao invés das duas, como já ilustrado no arquivo de configuração já citado no Capítulo 3. O arquivo de configuração foi instalado nos computadores intermediários *Click-1* e *2*, Figura 4.3.

Como se pode observar na Figura 4.3 a máquina denominada Servidor gera pacotes do tipo IP. Esses pacotes ao chegarem à máquina Click-1 são analisados pelo roteador IP construído com os elementos do *Click Modular Router* em Click-1, sendo descartados ou prosseguindo para o próximo nó, no caso, a máquina Click-2. Na máquina Click-2, os pacotes são agora analisados pelo roteador IP construído em Click-2 podendo ser enviados para as máquinas Click-3 ou Click-4 ou mesmo descartados. Os diagramas de blocos dos arquivos de configuração dos roteadores IP baseados em *Click Modular Router* das máquinas Click-1 e 2 encontram-se ilustrados a baixo das mesmas máquinas. O diagrama da máquina Click-1 usa duas interfaces de rede para entrada e saída dos pacotes, já a máquina Click-3 faz uso de três interfaces de rede também para entrada e saída de pacotes. Esses diagramas possuem o mesmo funcionamento do diagrama ilustrado e descrito na Figura 4.3.

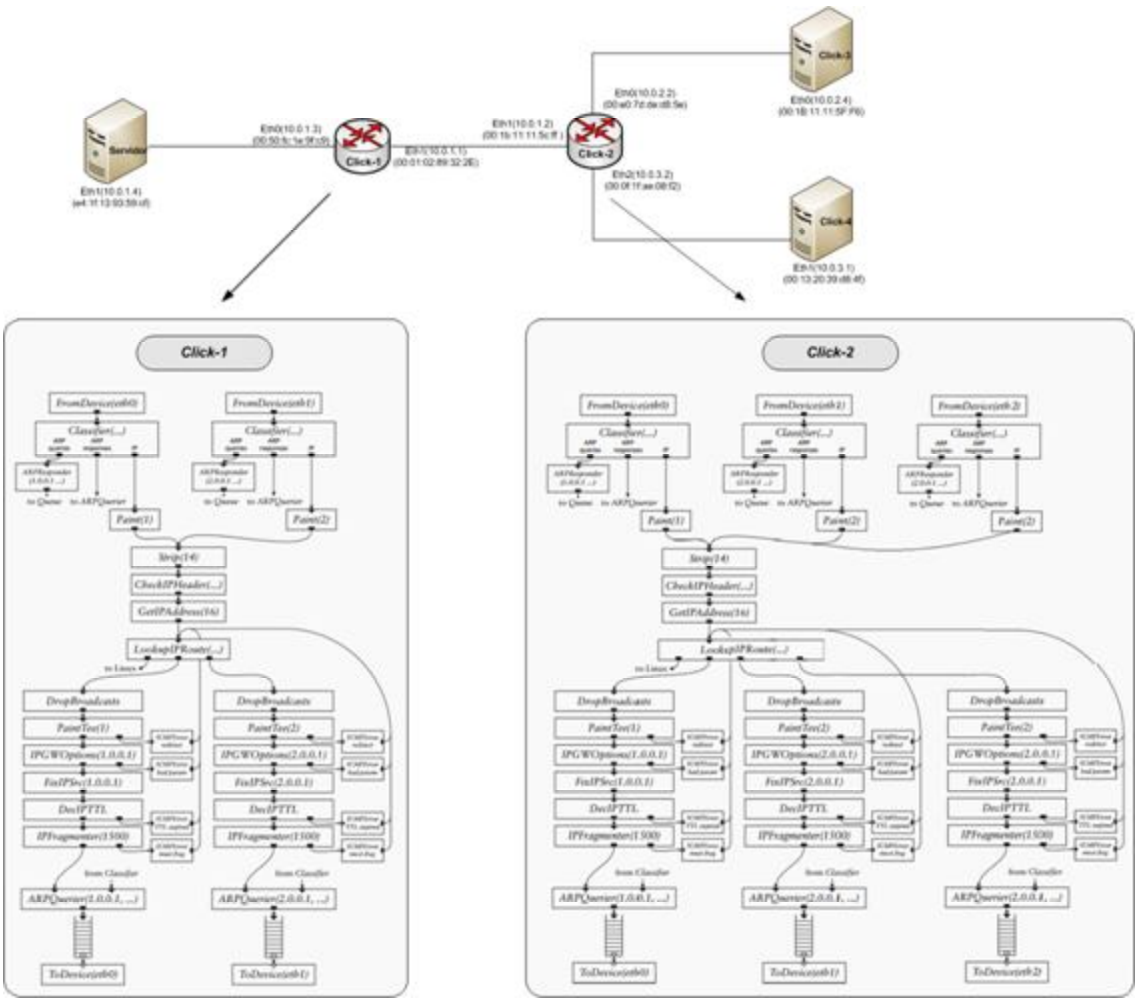

Figura 4.3 - Configuração Roteador Click Modular Router.

4.3.3 Encaminhamento, *Kernel Bridge*

Como forma de observar o desempenho do método de encaminhamento via *Kernel Bridge*, usou-se a mesma montagem física ilustrada na Figura 4.4, configurando cada uma das máquinas intermediárias para operarem como *bridges*. Para tanto, foi necessária a instalação do pacote *bridge* do Linux.

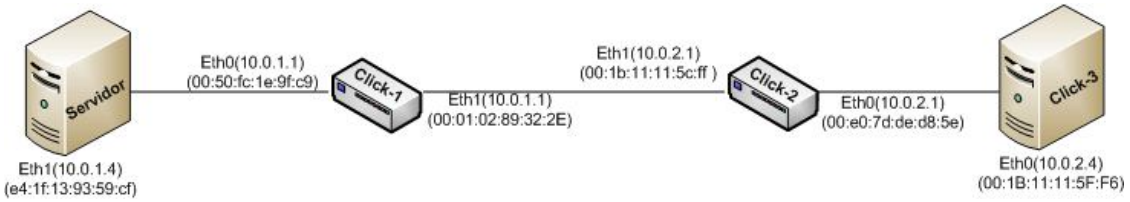

Figura 4.4 - Encaminhamento com Kernel Bridge.

4.3.4 Encaminhamento, *Click Modular Router*

Para a montagem do ambiente para encaminhamento utilizando o *Click Modular Router*, fisicamente no encaminhamento pacote a pacote, foi utilizada a seguinte montagem, ilustrada na Figura 4.5. A diferença em relação à montagem do encaminhamento via *kernel bridge* é que agora as máquinas intermediárias se encontram configuradas para funcionarem como comutadores implementados utilizando *Click Modular Router* em *kernel-module* e não mais com *bridge kernel*.

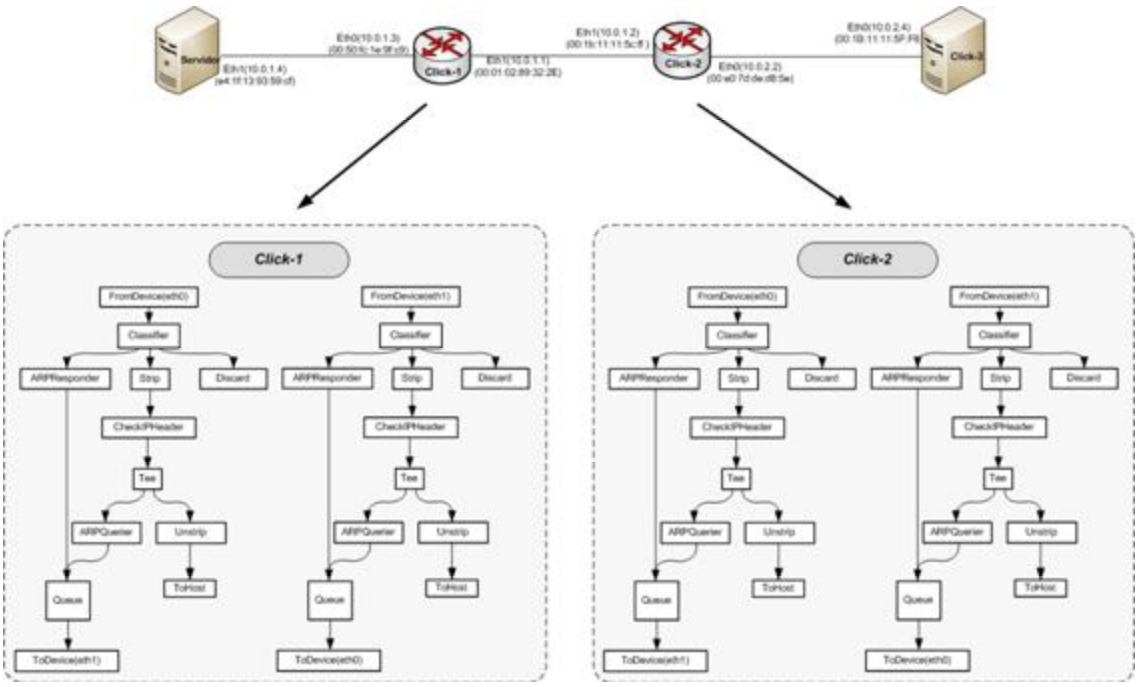

Figura 4.5 - Encaminha de pacotes usando *Click Modular Router*.

Para a implementação de encaminhamento baseado em rajadas (*bursts*) foi montado o *setup* ilustrado na Figura 4.6, onde existe o acúmulo dos pacotes e posterior envio dos mesmos na forma de rajadas.

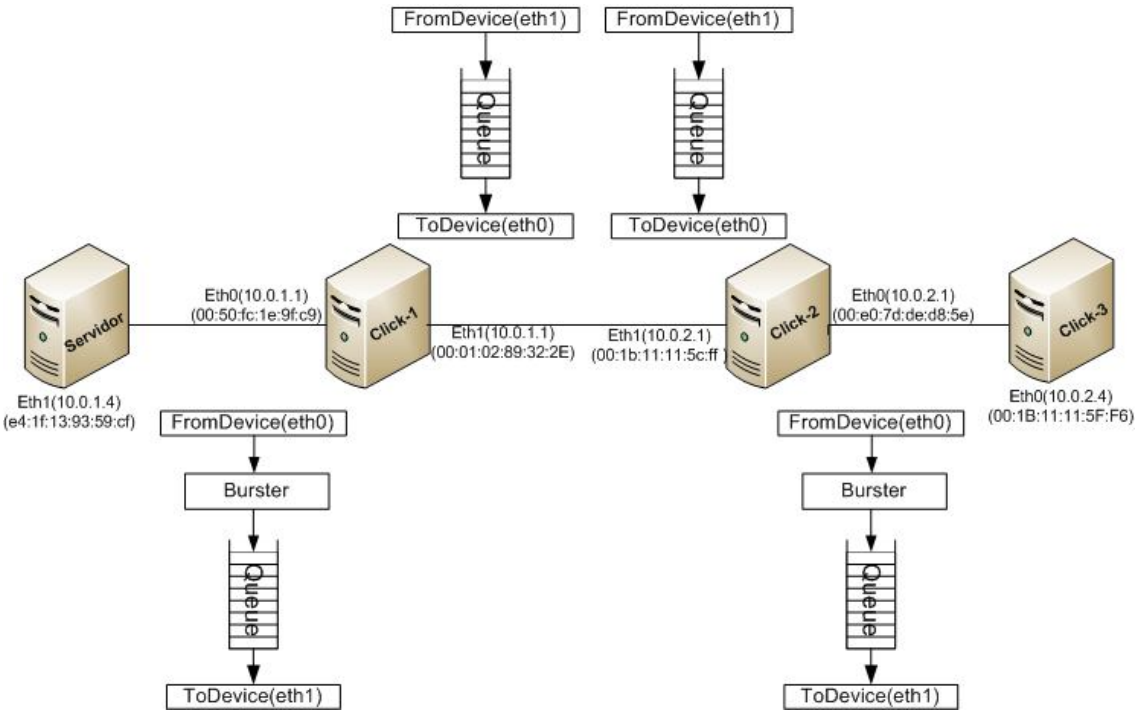

Figura 4.6 – Montagem e encaminhamento de rajadas (*bursts*) usando *Click Modular Router*.

4.3.5 Encaminhamento, uso do *OpenVswitch*

De forma a ampliar a quantidade de ambientes para análise e comparação de desempenho, também foi montado um ambiente com *OpenVswitch* [32]. Novamente foi usada a mesma configuração física descrita anteriormente, surgindo como diferencial neste caso o uso do *OpenVswitch*, atuando basicamente como encaminhador, nas máquinas *Click-1* e *2*, como pode observar em Figura 4.7.

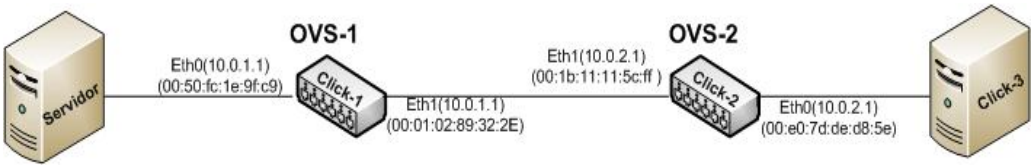

Figura 4.7 – Encaminhamento de pacotes com *OpenVSwitch*.

4.3.6 Transmissão via circuitos

Em laboratório também formam realizados testes envolvendo a transmissão de dados via circuitos. Os *setups* montados para a transmissão via circuitos são semelhantes a outros já ilustrados pelas Figura 4.4 e 4.6. A diferença encontra-se basicamente no fato de que o fluxo de dados que entra por uma interface de rede, após passar por um *buffer* é imediatamente encaminhado para a interface de saída sem que haja qualquer tipo de processamento. O *setup* que ilustra a implementação da transmissão via circuitos encontra-se ilustrado na Figura 4.8.

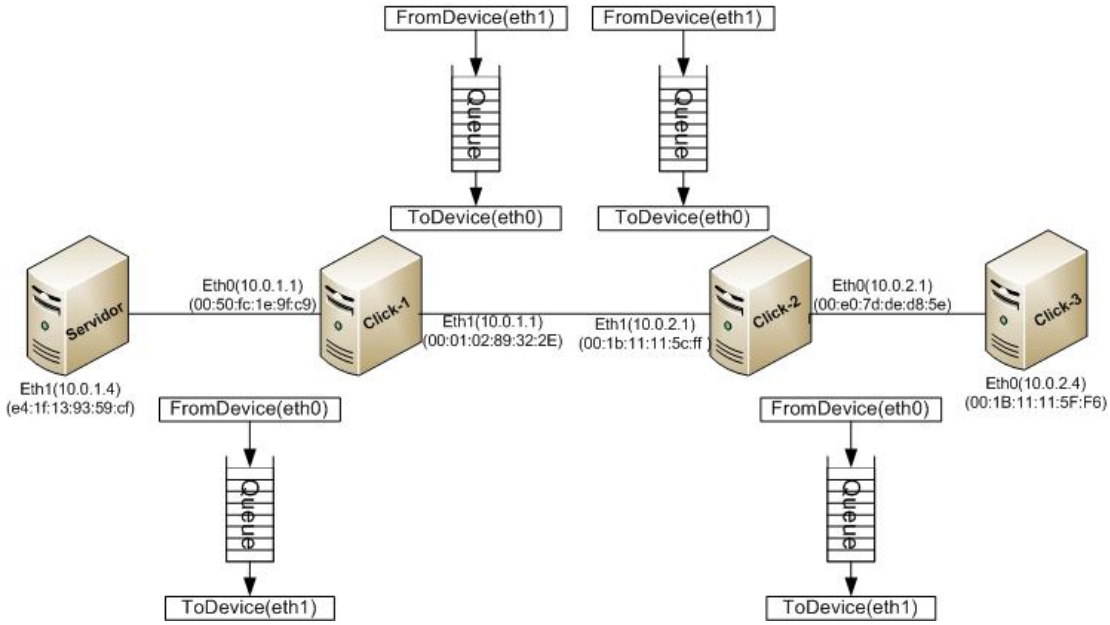

Figura 4.8 - implementação de circuito usando *Click Modular Router*.

4.4 Aferição de latência utilizando o *Oflops*

A fim de se adquirir medidas mais precisas foi utilizado o *framework Oflops* [33][34] associado a uma placa NetFPGA (Apêndice A1), com quatro interface com capacidade de 1 Gbps cada. A montagem de testes encontra-se ilustrada na Figura 4.9, onde se tem uma máquina operando com CentOS 5 e Oflops associado a NetFPGA. Por padrão, o pacote *actiondelay* do *Oflops*, necessita utilizar três interfaces de rede, pois duas se destinam à transmissão e recepção de pacotes e a terceira à coleta de estatíticas. Portanto pode-se notar pela Figura 4.9, a existência de três conexões entre as máquinas *Oflops+NetFPGA* e *Click-2*, sendo duas para o envio e recepção de dados, e uma de controle para coleta das estatísticas geradas.

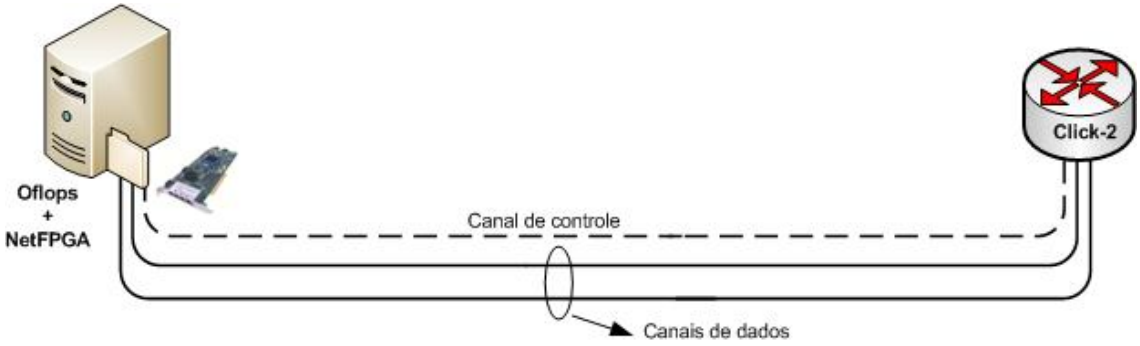

Figura 4.9 - Uso do *Oflops*/NetFpga.

4.4.1 Aferição de latência da conversão Eletro-Óptico com o Oflops

Como forma de observar a resposta de um sistema quando existe a conversão do domínio elétrico para o óptico e vice-versa, foram montados em bancada os experimentos ilustrados pelas Figura 4.10 e 4.11.

Na primeira montagem está ilustrado o teste de *loopback* totalmente em meio elétrico, onde as quatro interfaces da NetFPGA, encontram-se em curto-circuito duas a duas, formando-se os canais de dados e de controle, assim como ilustrado em Figura 4.10.

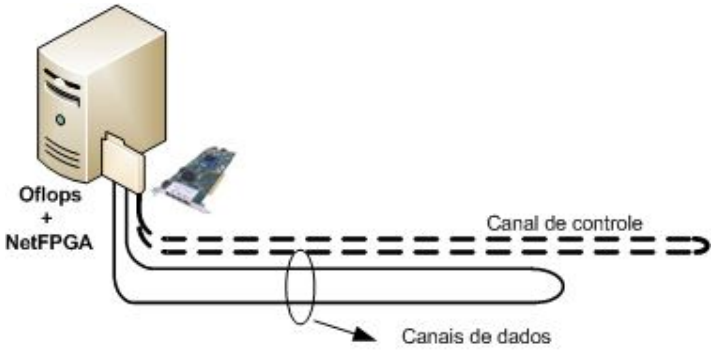

Figura 4.10 - Setup Loopback Oflops.

Já na Figura 4.11 o cenário é quase idêntico ao anterior, exceto pela existência nos canais de dados de conversores de mídia, cuja função é a de realizar a conversão dos sinais elétricos gerados e transmitidos pelo Oflops + NetFPGA para o domínio óptico e posteriormente realizar a conversão reversa.

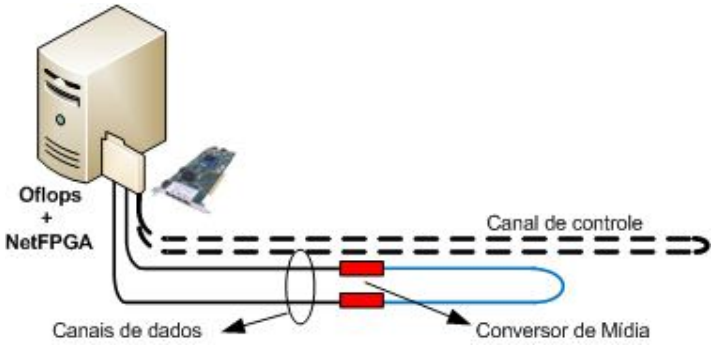

Figura 4.11 - Setup Loopback *oflops* com conversor de mídia.

Os resultados das aferições realizadas em ambos os testes encontram-se no capítulo que trata dos resultados atingidos.

4.5 Montagem experimental de um ambiente com rajadas

Em laboratório foi montado um ambiente para transmissão de dados na forma de rajadas, como ilustrado pela Figura 4.12. O ambiente é composto por cinco elementos, sendo um responsável pela autorização da transmissão das rajadas e pela configuração dos demais elementos. A máquina denominada *"servidor"* é um computador emulando duas máquinas virtuais responsáveis por gerar tráfego, a máquina *"Click-3"* é o destino do tráfego gerado pelo *"servidor"*, já as máquinas *"Click-1 e 2"* atuam como comutadores.

No ambiente montado em laboratório e ilustrado pelo Figura 4.12, se tem a solicitação de comunicação por parte da máquina *servidor* com a máquina *Click-3* por via da troca de mensagens de sinalização com uma máquina externa, responsável por arbitrar o acesso ao meio de transmissão, definir e configurar o caminho por onde as rajadas de dados iram trafegar.

Na Figura 4.12 os números que vão de 1 a 10 ilustram a sequência de mensagens de configuração trocadas e também citam o tipo de mensagem. Já as setas dispostas na Figura 4.12 informam a origem e o destino de cada uma das mensagens de configuração utilizadas.

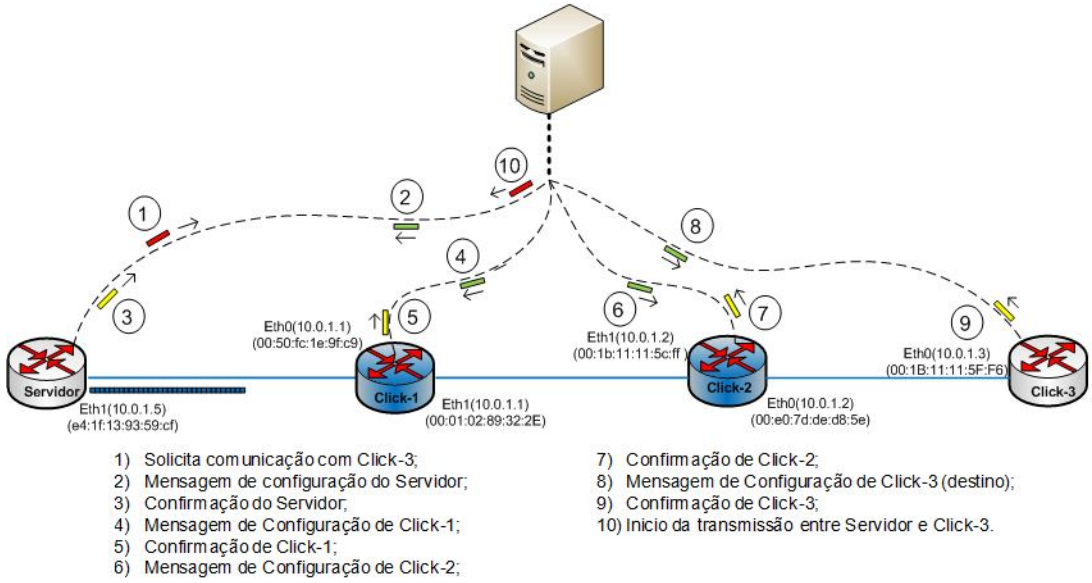

Figura 4.12 - Ambiente de transmissão por rajadas.

A finalidade de tal experimento está na possibilidade de construção de comutadores baseados na API *Click Modular Router*, além da visualização da limitação imposta por um ambiente não transparente, ou seja, ambiente com a conversão E/O/E em oposição à comutação totalmente em meio óptico. Outro objetivo do setup é propriamente a construção de um ambiente que funcione segundo a transmissão por rajadas de pacotes. Como a configuração dos comutadores, *click-1* e *2*, é feita imediatamente após a recepção e processamento do quadro de controle havendo um intervalo de tempo entre o quadro de controle e a rajada, o modelo implementado segue a ideia do JIT (*Just In Time)*, apêndice A2.

4.6 Sumário

Neste capítulo foram descritos e ilustrados todos os experimentos realizados, além da justificativa de suas montagens. O objetivo foi explicar de forma clara todos os procedimentos realizados, visando uma melhor interpretação dos resultados apresentados no capítulo que se segue.

Capítulo 5: Resultados

No capítulo anterior foram apresentados todos os *setups* utilizados durante a realização dos testes experimentais em laboratório, descrevendo e justificando cada um dos testes. Neste capítulo, pretende-se apresentar uma análise comparativa do impacto que alguns modelos de transmissão de dados podem causar sobre uma rede de *datacenter* centrada em servidores. Os parâmetros tomados como base são: latência, taxa de transmissão e perda de quadros. Observar-se-á que tais parâmetros aferidos se encontram associados ao consumo de CPU dos servidores quando desempenhando atividades de comutadores/encaminhadores de tráfego em detrimento de atividades exclusivas do servidor. De forma a tornar mais flexível a realização de tal estudo foi usada uma plataforma aberta para implementação, *Click Modular Router*, já descrita no capítulo 3.

5.1 Latência na conversão Eletro-Óptica

A fim de mensurar limitações impostas pela conversão do domínio óptico para elétrico, uma vez que todos os dados coletados e discutidos no trabalho são referentes a comutadores que atuam no domínio elétrico, foram realizados os experimentos de latência já descritos no capítulo 4 e ilustrados nas Figura 4.10 e 4.9. Os resultados se encontram ilustrados nas Figura 5.1 e 5.2.

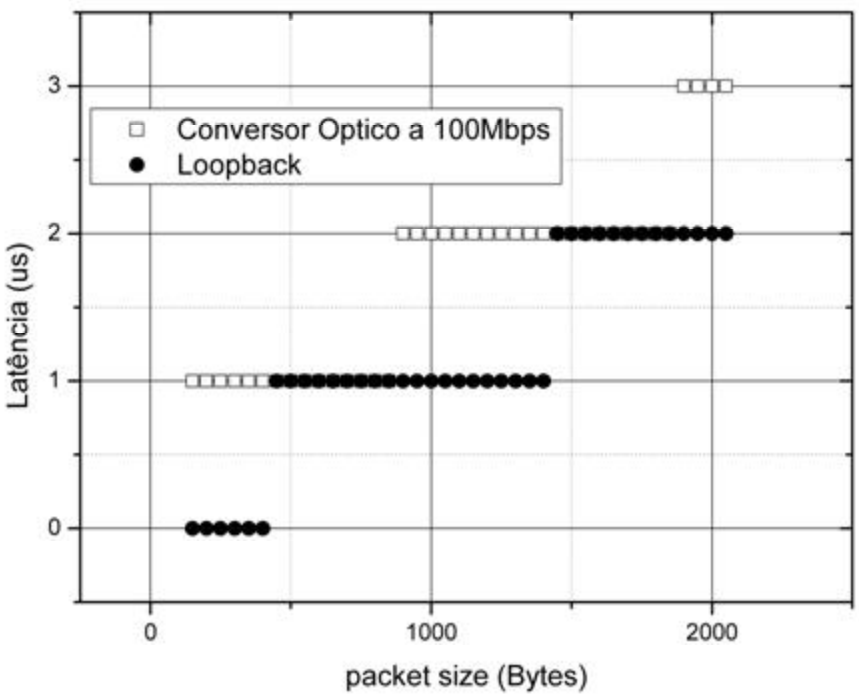

Figura 5.1 - Conversor Óptico vs Loopback Elétrico, taxa de 100Mbps com variação do tamanho dos quadros.

Visivelmente nota-se que embora seja mínima a diferença de respostas entre o sistema de *loopback* totalmente elétrico e o com conversores ópticos, tal diferença está presente. O sistema com o conversor óptico apresenta uma maior latência para pacotes com maior tamanho em comparação com o sistema sem conversor, como ilustrado no gráfico da Figura 5.1. Para a situação onde o tamanho do pacote é constante e a taxa é variada, Figura 5.2, o *loopback* elétrico e o com conversor de mídia respondem de forma idêntica em grande parte do experimento, só apresentando uma latência superior por parte do conversor de mídia para taxas de dados próximas de 1Gbps, como pode-se observar na Figura 5.2.

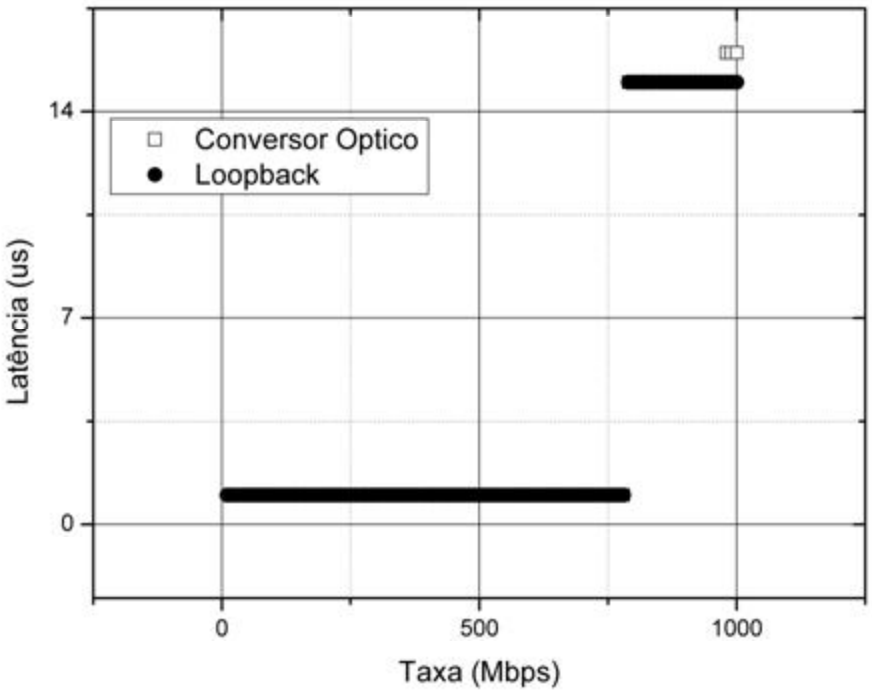

Figura 5.2 - Comparação Conversor Óptico vs Loopback Elétrico, tamanho de quadro de 250 bytes e taxa variável entre 10 e 1000 Mbps.

Com os resultados ilustrados nas Figura 5.1 e 5.2, podemos observar que a existência da conversão eletro-óptica gera um aumento da latência no encaminhamento de pacotes. O aumento da latência é particularmente nítido na situação em que o fluxo de dados é composto por pacotes com tamanho crescente. Em se tratando de poucos nós a latência é pequena, no entanto quando se tem grandes redes quanto à quantidade de nós podem-se atingir valores bem significativos de latência acumulada.

5.2 Ocupação do processador por parte do Click

As Figura 5.3 e 5.4 ilustram uma comparação quanto ao consumo de CPU de um procedimento repetitivo de instalação/desinstalação, laço de 100000 instalações, seguido de desinstalação (*click-install* e *click-uninstall*) de arquivos de configuração no *Click Modular*

Router em oposição ao processo repetitivo de instalação usando o *hotswap*, ferramenta presente no Click. O objetivo é observar até que parcela do sistema pode ser comprometida pelo simples fato de carregar os módulos do *Click Modular Router*. O teste torna possível obter uma relação de quanto é o consumo apenas da execução do Click Modular Router e de quanto é o consumido apenas na comutação de dados, possibilitando uma estimativa de qual a disponibilidade da CPU no processamento de outras atividades.

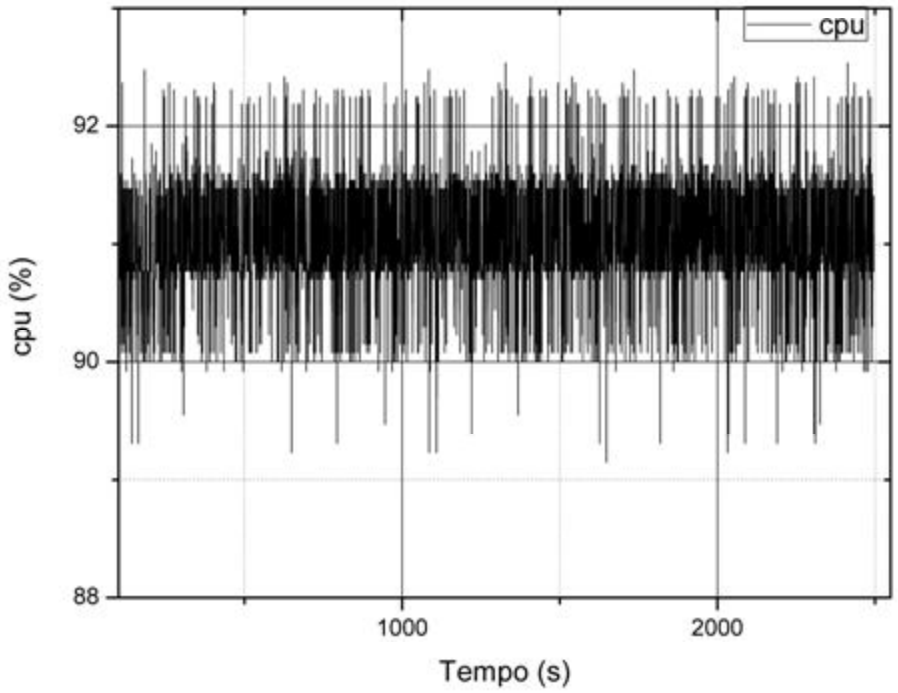

Figura 5.3 - Consumo de CPU sem o uso do *hotswap*.

Com a Figura 5.3 se pode notar que sem o uso da ferramenta *hotswap* no processo de atualização e instalação dos arquivos de configuração do *Click Modular Router* a disponibilidade da CPU é menor que 10% para o encaminhamento de dados e execução de qualquer outra tarefa.

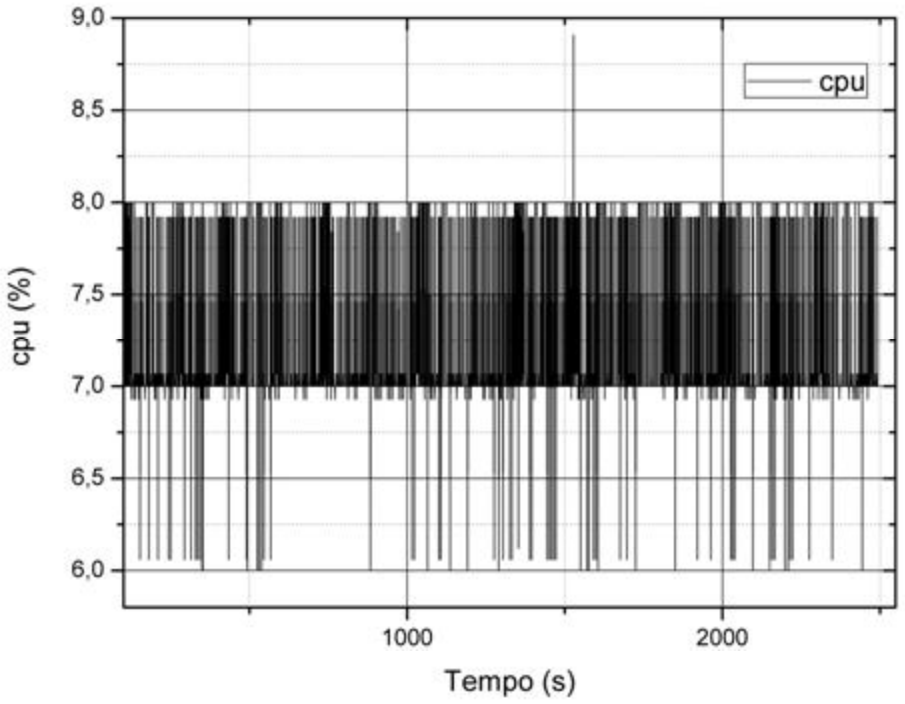

Figura 5.4 - Consumo de CPU com o uso do *hotswap*.

Nota-se que apenas com o uso da ferramenta *hotswap* do *Click Modular Router*, o processo de instalação do mesmo passa a impor um menor consumo de CPU, girando entre 7 e 8%, quando comparado com o processo de instalação seguido de desinstalação, que consome recurso superior a 90%. Portanto, quando se passa a usar a ferramenta *hotswap* no processo de modificação do arquivo de configuração tem-se uma grande economia do consumo da CPU quando comparado o processo de desinstalação e reinstalação do mesmo arquivo no *Click Modular Router*, chegando a uma relação de 13 vezes menor.

O ganho de eficiência adquirido com o uso da ferramenta *hotswap* dá-se pelo fato de a mesma quando no ato da instalação de um novo arquivo de configuração, atua carregando apenas os módulos, elementos, que não estavam presentes no arquivo de configuração anteriormente em execução. Quando não é usada a ferramenta *hotswap*, faz-se a desinstalação de todos os módulos, seguida da reinstalação.

5.3 Desempenho do servidor como encaminhador

Nesta seção serão ilustrados e discutidos os resultados obtidos via testes em laboratório dos *setups* já descritos anteriormente em seções deste mesmo capítulo. Os resultados aqui mencionados são referentes à latência média, consumo de CPU e perda de pacotes. O objetivo é mensurar o quanto algumas formas de ocupação do meio de transmissão de dados podem afetar os comutadores intermediários e por consequência ser prejudicada pelos mesmos.

5.3.1 Taxa de transmissão

Quanto à taxa de transmissão de dados foram realizados testes usando o software *iperf* para criar e gerenciar conexões TCP e UDP entre máquinas virtuais *Click-6* e *7* presentes na máquina *servidor*, e máquina *Click-3*, passando por *Click-1* e *2*, Figura 5.5.

Das conexões estabelecidas via *iperf*, foram coletados medidas de taxa de transmissão para um ambiente operando com roteador IP e rajada (*burst switch*), ambos implementados com *Click Modular Router*. Também foi realizando utilizando o encaminhamento via *Linux Kernel*, como ilustrada na Figura 4.1.

O retângulo pontilhado representa uma máquina física emulando duas máquinas virtuais, *Click-6* e *7*.

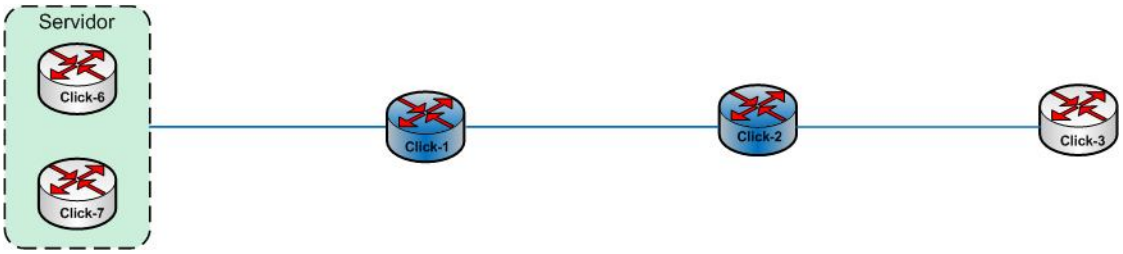

Figura 5.5 - Setup com máquinas virtuais no servidor.

No roteador IP implementado com *Click Modular Router, clickIP,* foi usado o mesmo arquivo de configuração fornecido como exemplo pelo desenvolvedor, com apenas algumas adaptações já comentadas anteriormente. Para a implementação com encaminhamento por rajadas, também em *Click Modular Router*, foram montadas rajadas com 10000 pacotes e intervalo de 0,005s entre rajadas. Os parâmetros do tamanho das rajadas e do intervalo de tempo entre as mesmas foram escolhidos de forma aleatória, escolhendo os valores de 10000 pacotes e 0,005s.

Com os resultados apresentados na Figura 5.6, percebe-se que para o click-OBS e para Kernel Linux, os resultados são praticamente idênticos, embora o primeiro tenha sido montado totalmente usando Click em *kernel-module* e o segundo diretamente no Kernel do sistema operacional. Já no caso da construção do roteador IP no Click, tem-se que o desempenho se apresenta bem inferior quando comparado com as duas construções anteriores. Portanto, mesmo não sendo implementado diretamente no Kernel, mas em outra aplicação executada sobre o mesmo, o método operando com rajadas apresenta resultado mais satisfatório que o roteador IP em se tratando de taxa discreta de dados transmitida atingida, percebe-se também uma grande similaridade entre o clickOBS e o roteamento realizado diretamente pelo Linux ou Kernel Linux.

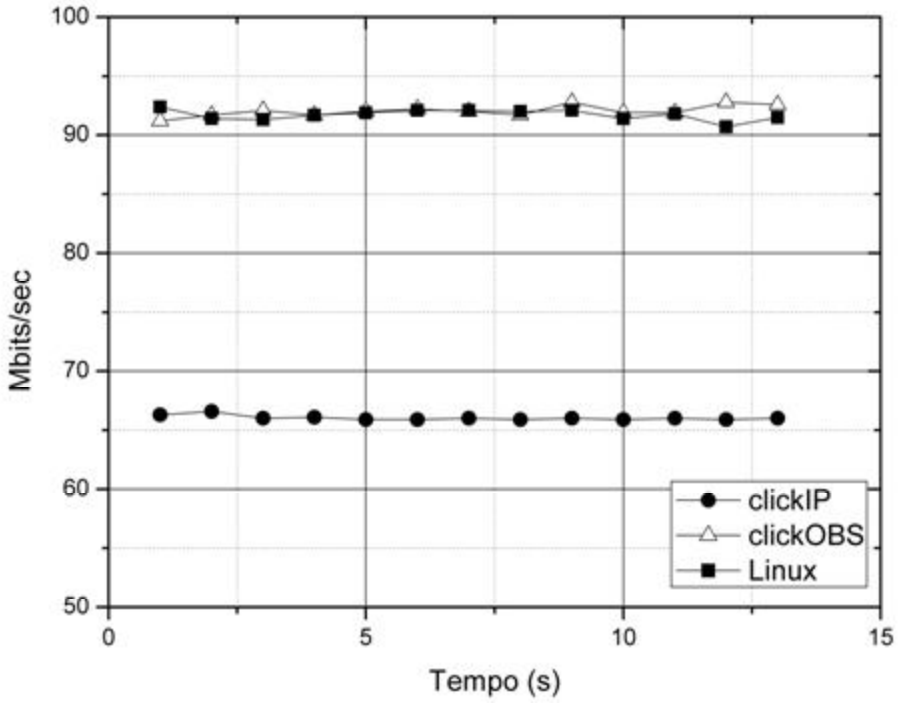

Figura 5.6 – Comparação entre a taxa de transmissão atingida com a implementação do roteador IP no Click e diretamente no Kernel do Linux e encaminhador de rajadas no Click.

Por via de se compreender melhor os resultados apresentados, foram realizadas simultaneamente aos testes de vazão, medidas de ocupação de CPU em ambas as máquinas intermediárias, *Click-1* e *2*, sendo ilustrados nas Figura 5.7 e 5.8.

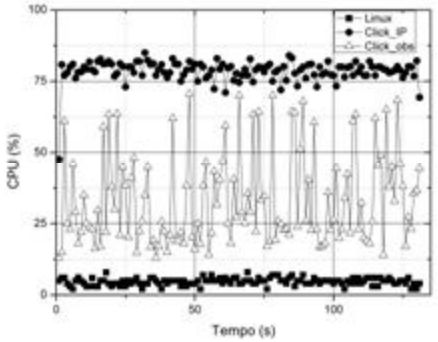

Figura 5.7 - Comparação de desempenho quanto a o uso de CPU da máquina Click-01 para roteamento IP com Click, encaminhamento OBS Click e roteamento linux.

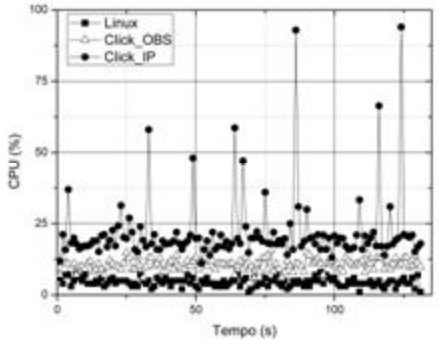

Figura 5.8 - Comparação de desempenho quanto a o uso de CPU da máquina Click-02 entre roteamento IP com Click, encaminhamento OBS Click e roteamento Linux.

Como se pode observar, tem-se que o consumo de CPU em ambas as máquinas para o roteamento IP implementado em Click é superior às demais implementações, inclusive quando comparada a com o uso de rajadas associadas ao *Click Modular Router*, o que provavelmente se reflete no resultado da taxa de transmissão de dados. O fato de alto consumo de CPU nas duas máquinas tem relação com o fato das mesmas serem máquinas antigas, com processadores Pentium IV, 1024 Mbytes de memória, o que acaba por evidenciar o alto consumo de recursos. Pelas Figura 5.7 e 5.8, nota-se que existe um maior consumo da CPU da máquina Click-1 quando comparado à máquina Click-2. Tal acontecimento se dá, pois sendo um fluxo unidirecional entre as máquinas, a máquina Click-1 acaba por se tornar um gargalo ao sistema, uma vez que tendo um alto índice de ocupação a mesma tem uma alta porcentagem de perdas de pacotes, reduzindo por consequência a taxa de dados que chegam a máquina Click-2, que por sua vez acaba atingindo uma menor ocupação de CPU.

Tomando-se por base a taxa de vazão e o consumo de hardware, Figura 5.6, 5.7 e 5.8 pode-se reforçar a ideia de que, se bem administrada, a forma de alocação do meio com o uso de rajadas traz certa vantagem, pois atinge altas taxas de dados associado a uma menor ocupação de hardware quando comparada ao modelo equivalente baseado em roteamento IP também construído com o *Click Modular Router*. Portanto a comutação por rajadas possui vantagens quando em comparação com o roteamento em IP quanto à taxa de pacotes atingida

e quanto ao consumo de CPU. Tal comparação foi possível com a implementação do comutador por rajadas e do roteador IP ambos no *Click Modular Router*.

Com relação à variação do tamanho dos quadros sendo associados em rajadas ou não, temos que o índice de ocupação da CPU se apresenta com pouca variação, como pode ser visto nas Figura 5.9 e 5.10, o que ilustra que a variação do tamanho dos pacotes não causa sobrecarga extra na CPU dos comutadores. Nestes testes foi feita a comparação, com a variação do tamanho dos pacotes de 64 a 1500 bytes, dos modos de transmissão sem o uso do *Click*, no caso realizando o encaminhamento diretamente via o Kernel do Linux. Os outros testes foram feito com o *Click*, atuando como encaminhador de pacotes, atuando como encaminhador de rajadas com 5ms e 0,5ms de intervalo entre rajadas.

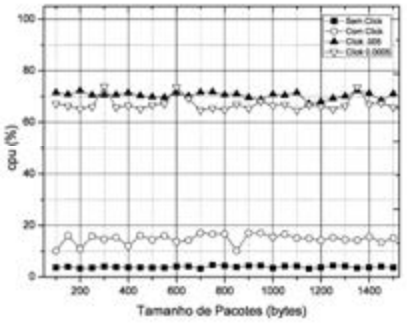

Figura 5.9 - Ocupação de CPU Click-2 por tamanho de pacote.

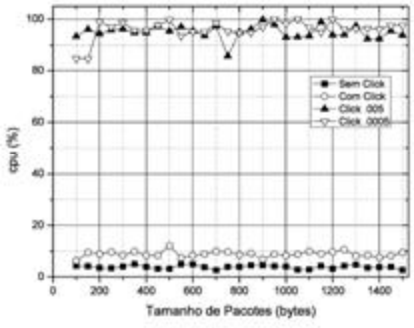

Figura 5.10 - Ocupação de CPU Click-1 por tamanho de pacote.

5.3.2 Latência

Para os testes de latência foi utilizado o *framework Oflops* associado à NetFPGA, visto que o mesmo possibilita uma maior precisão, chegando a ordem de microssegundos, ver apêndice A1. Devido ao fato do *Oflops* ter sido desenvolvido para realizar testes de desempenho em redes *OpenFlow*, seu método de geração e transmissão de dados se baseia em pacotes.

De forma a enriquecer a análise comparativa, foram feitas medidas utilizando-se outras ferramentas de encaminhamento de dados, como *Kernel Bridge*, encaminhamento associado a regras estabelecidas via *Ebtable*, *Bridge* estabelecida sobre *OpenVswitch* e encaminhador criado com *Click Modular Router*, sendo o resultado apresentado na Figura

5.11. Todos os resultados apresentados por Figura 5.11, 5.12 e 5.13 são referentes à comutação por circuito, pois como já mencionado, os módulos do *Oflops* realizam toda a geração e transmissão de dados baseado em pacotes. Portanto as medidas de latência e perda de pacotes na transmissão por rajadas não apresentariam resultados reais, sendo, no caso da latência, mascarada pelos intervalos entre rajadas e na perda de pacotes, impossibilitada pelo próprio processo de geração e transmissão padrão do *Oflops*. Com isso optou-se por realizar uma comparação entre as ferramentas de encaminhamento de dados, implementando encaminhamento via comutação de circuitos em todas as ferramentas citadas, *Kernel Bridge*, *Ebtable*, *OpenVswitch* e *Click*. As implementações com comutação por pacotes não foram realizadas.

Enquanto todos os testes estão relacionados a uma máquina intermediária, onde estão implementados os encaminhadores, no *loopback* trata-se de um "curto-circuito" entre as interfaces da NetFPGA, a fim de ilustrar a latência imposta pela própria ferramenta utilizada, NetFPGA + *Oflops*, que como pode-se perceber é mínima.

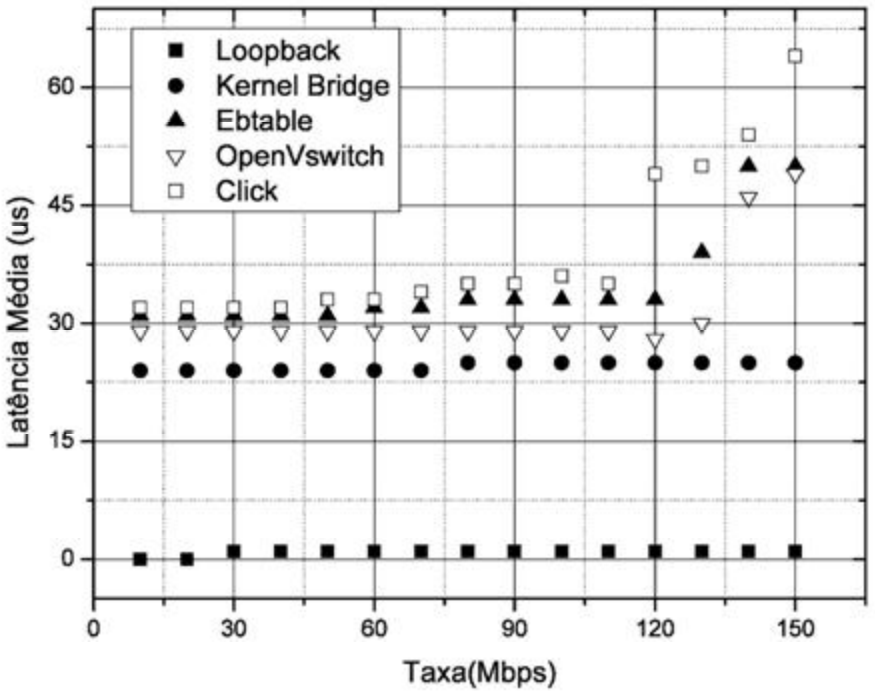

Figura 5.11 - Latência média por taxa de transmissão para pacotes de 250 bytes.

Como pode ser observado pela Figura 5.11, não há grande alteração na latência imposta pelos métodos de encaminhamento testados até a taxa de 120 Mbps, sendo o feito pela *Bridge* diretamente construída em Kernel Linux a de menor latência, provavelmente devido ao fato de as demais implementações serem executadas sobre o próprio Kernel. Para taxas a cima de 120 Mbps a latência aumenta razoavelmente quando comparada a imposta pelo *Kernel Bridge*, o que pode está associado ao uso de máquinas antigas com hardware limitado que demonstraram sofrer uma alta ocupação de CPU, e por consequência embora possuam NIC com capacidade de 1Gbps, o intenso consumo de processamento acaba refletindo negativamente na forma de uma crescente latência.

Além da latência em função do incremento da vazão, também foram realizadas aferições que ilustram o comportamento da latência com o incremento do tamanho dos quadros para uma vazão constante, como ilustrado na Figura 5.12.

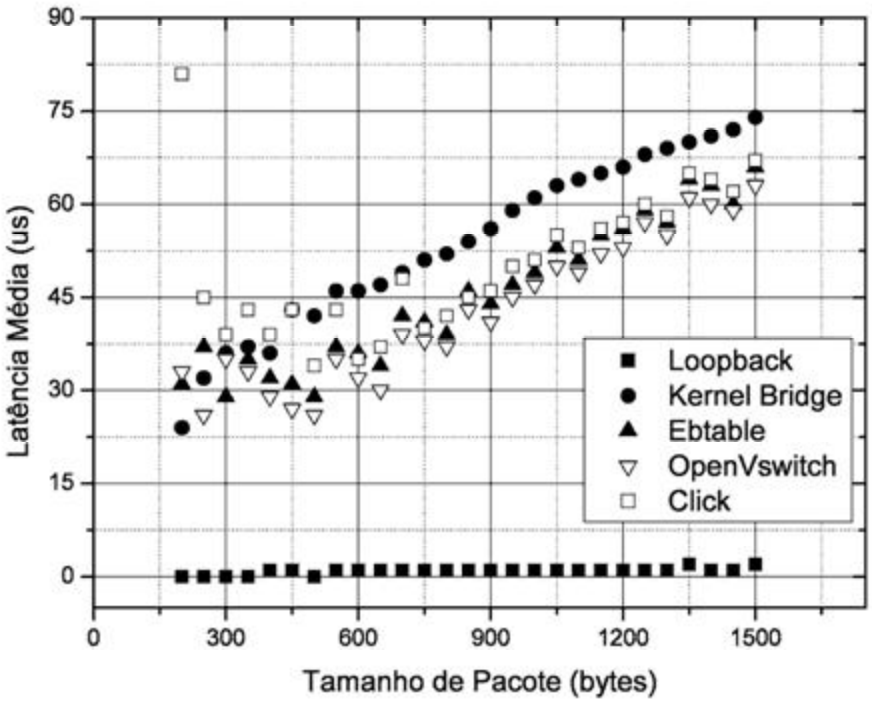

Figura 5.12 - Latência média por tamanho de pacotes a 100 Mbps.

Algo de fácil percepção é que em todos os casos testados, para pacotes de tamanho superior a 512 bytes, há uma relação próxima da linear entre a latência e o tamanho dos pacotes, o crescimento de um leva ao do segundo, já no caso de pacotes de menor tamanho a relação não se faz mais presente, havendo momentos em que a latência imposta por uma

implementação é menor que a de outra, e em outros momentos onde a situação se inverte. Outra observação importante é que o Kernel Bridge tem um desempenho relativamente inferior aos demais quando lida com quadros maiores que 512 bytes, provavelmente ocasionado pela fragmentação dos quadros.

5.3.3 Perdas de pacotes

Um novo teste feito, também baseado no encaminhamento por circuitos, foi o de analisar o comportamento de cada um dos métodos de encaminhamento novamente com a variação da taxa de dados com tamanho de quadros e relacionando com a perda dos mesmos.

Na Figura 5.13 nota-se uma alta porcentagem de perda de dados com o aumento da taxa de transmissão, inclusive no *loopback*. O que leva a clara limitação interna do hardware das máquinas usadas no ambiente de testes, visto que as mesmas apresentam altas porcentagens de perda de pacotes no decorrer do aumento da taxa de envio de dados.

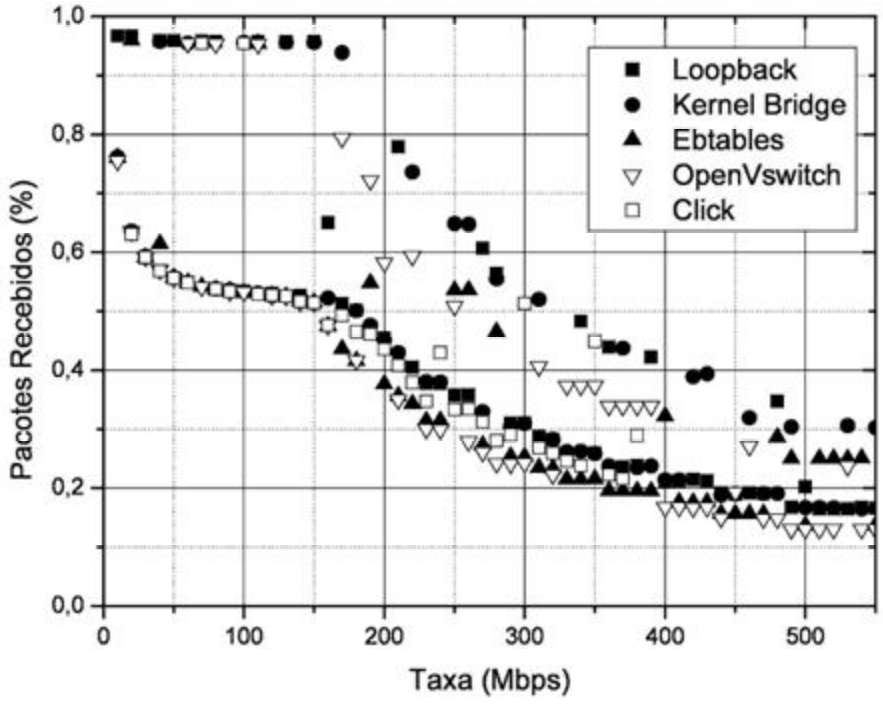

Figura 5.13 - Porcentagem de pacotes recebidos por taxa de transmissão.

Pela Figura 5.13 pode-se observar que com a taxa de dados variando entre 10 e 550Mbps, as perdas em função do tamanho de pacotes são grandes até mesmo para as medidas de *loopback*. O valores mensurados crescem a partir de valores inferiores a 10% de perda de pacotes, ou mais de 90% de pacotes recebidos, atingindo ordem de 50% de perdas para taxas de 150Mbps. Observa-se, quanto à taxa crescente de pacotes, todas as implementações se comportam de forma bem parecida, embora a implementação usando Kernel Bridge apresente uma pequena vantagem em relação as demais implementações, Figura 5.13.

Para os resultados obtidos com taxa de fixa em 100Mbps com tamanho dos pacotes variando entre 64 e 512 bytes, nota-se certa tendência da porcentagem da perda de pacotes permanecer em 50% e em alguns instante atingir porcentagens de perda inferior à 10%, Figura 5.14. Portanto o pode-se notar que a variação no tamanho dos pacotes não alterou o desempenho da rede quanto à taxa de pacotes recebidos.

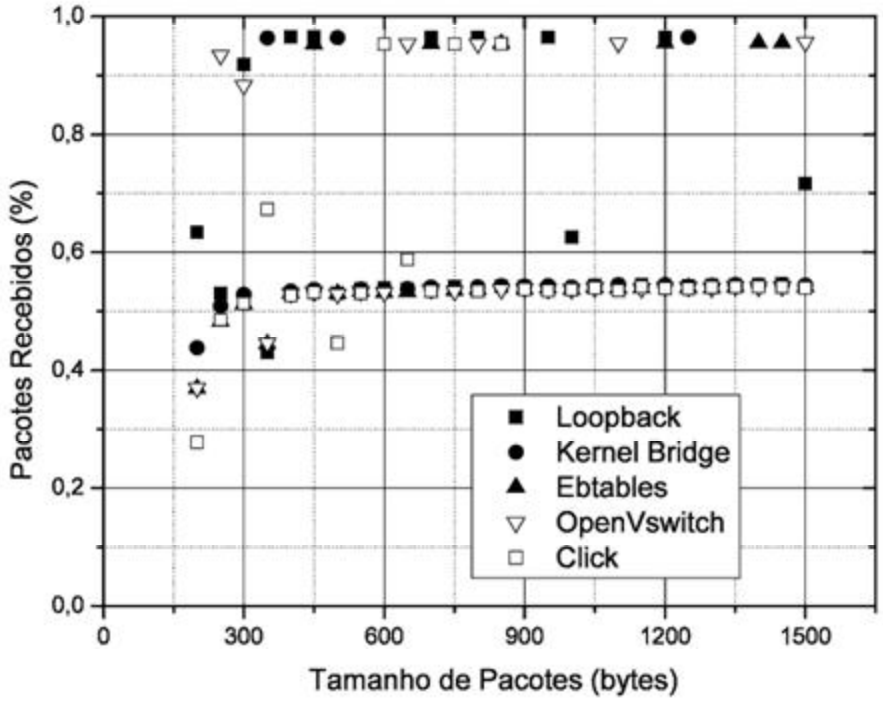

Figura 5.14 - Porcentagem de quadros recebidos por tamanho de quadro, Click implementando circuito.

5.4 Resultados com o setup de rajadas

A partir do *setup* ilustrado pela Figura 4.12 foram feitos testes de vazão por via da ferramenta *Iperf*. O experimento foi feito criando-se tráfego por via do *Iperf* e transmitindo-se os mesmos por via de rajadas, escolhidas ao acaso, de 1000, 10000 e 100000 pacotes. Pacotes estes com tamanhos que variavam de 100 a 500 bytes. A escolha de pacotes com tamanho variando de 100 a 500 bytes, assim com as rajadas com rajadas de 1000, 10000 e 100000 pacotes foram escolhidas ao acaso, não levando em consideração nenhum fator que pudesse influenciar os testes realizados. Na Figura 5.15, encontra-se representada à taxa média para as rajadas de 1000, 10000 e 100000 pacotes.

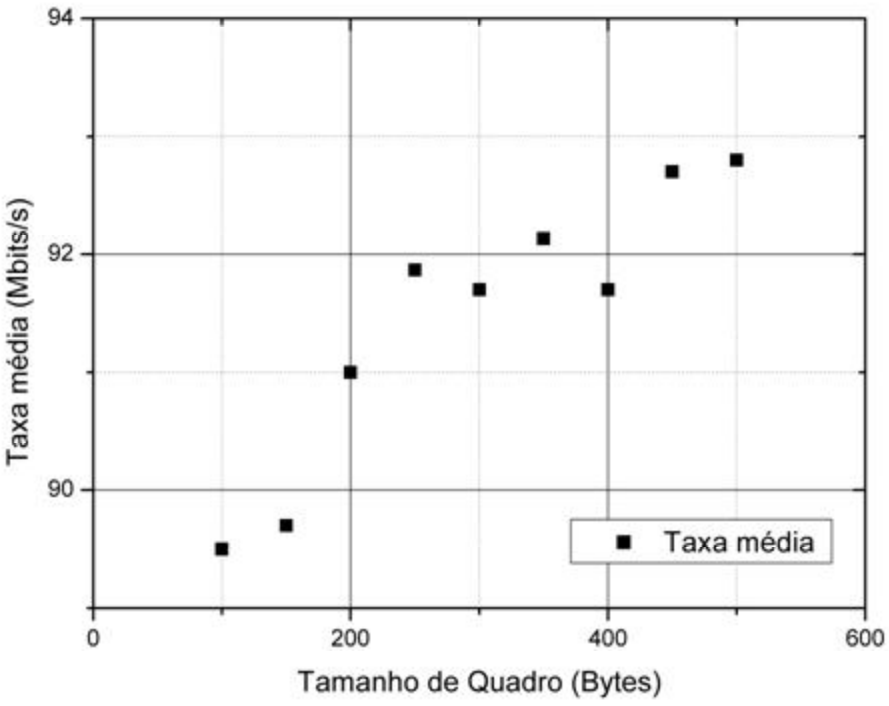

Figura 5.15 - Vazão entre Servidor e Click-2.

Durante o mesmo período em que se mediu a vazão, também foram coletados informações relativas à porcentagem de ocupação do processador das máquinas atuantes como comutador e destino do tráfego, Figura 5.16 e 5.17.

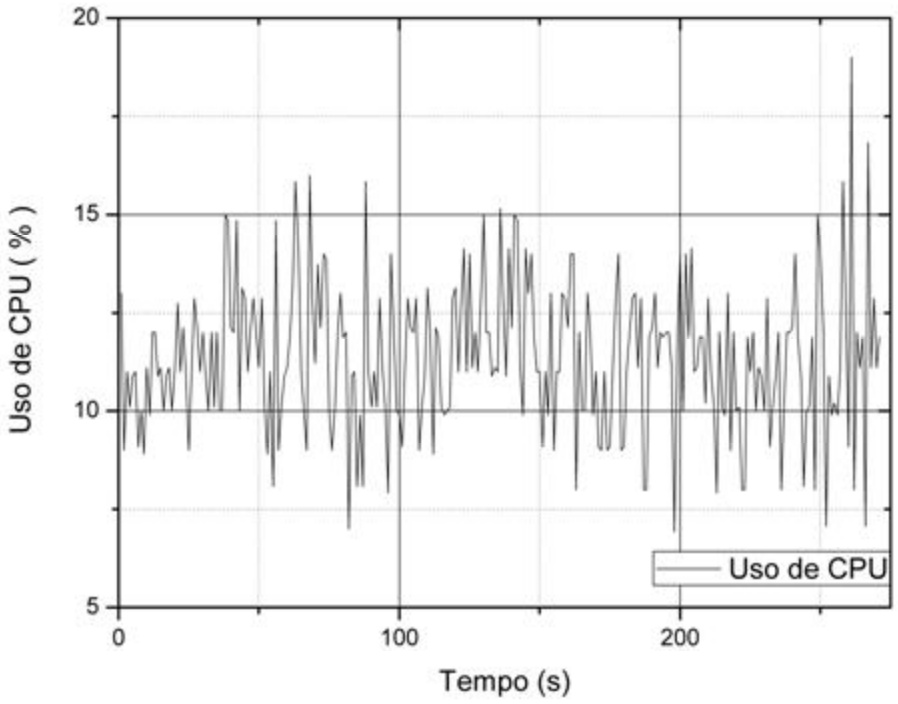

Figura 5.16 - Ocupação de CPU da máquina destino das rajadas: Click-2.

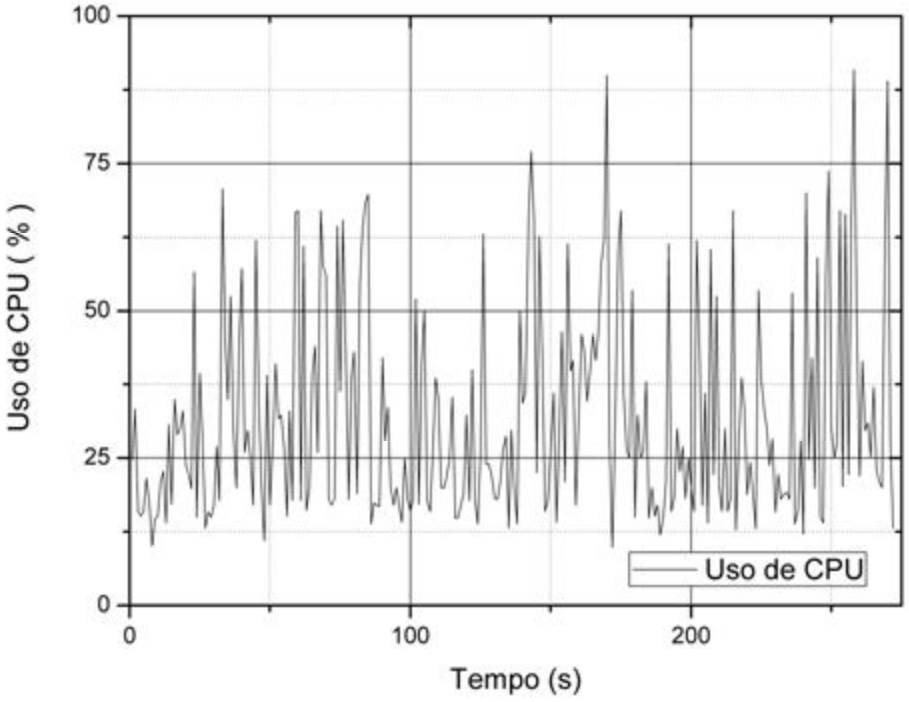

Figura 5.17 - Ocupação de CPU do nó comutador das rajadas: Click-1.

Novamente percebe-se a menor taxa de ocupação de CPU na máquina Click-1 quando em comparação a Click-2, ocorrendo, pois como já comentado anteriormente, a primeira máquina, Click-1, acaba por se tornar um gargalo ao sistema. Devido à taxa de perdas inseridas o fluxo que chega a segunda máquina é reduzido, refletindo no menor consumo da CPU da máquina Click-2.

5.5 Sumário

Neste capítulo foram apresentados dados coletados e discutidos a partir dos *setups* descritos e ilustrados no capítulo 4. Aqui foram feitas discussões relacionadas à latência, perda de pacotes, taxa de dados atingida e consumo de CPU, e como as mesmas se relacionam e podem influenciar no desempenho das trocas de dados tendo como um exemplo em especial à aplicação em uma rede *datacenter* centrada em servidores.

Capítulo 6: Conclusão e Trabalhos Futuros

Neste presente capítulo pretende-se apresentar uma discussão dos resultados apresentados ao longo dos capítulos anteriores, além de citar possíveis temas relacionados a este trabalho e que se pretende futuramente desenvolver um estudo aprofundado em outros trabalhos.

6.1 Contribuições

Como principal contribuição do presente trabalho teve-se a identificação e discussão de como as formas de transmissão de dados em uma rede pode influenciar no seu desempenho, uma vez que a ocupação de excessiva de CPU tem impacto sobre eficiência de repasse dos pacotes. As redes de *datacenter* centradas em servidores se encaixam perfeitamente no estudo aqui apresentado, pois nas mesmas o processo de manipulação ou encaminhamento do fluxo de dados é realizado pelos próprios servidores à custa do uso da capacidade de processamento.

Outra contribuição aqui apresentada foi a implementação do modo de transmissão por rajadas e por circuitos usando a ferramenta de desenvolvimento *Click Modular Router* e sua comparação com as formas nativas de encaminhamento do sistema operacional.

6.2 Conclusão

No decorrer do trabalho notou-se que se por um lado a manipulação, quanto ao encaminhamento do fluxo de dados, por computadores de uso geral agrega um ganho na flexibilidade do gerenciamento, associando maior inteligência a rede, por outro lado exige-se dos computadores uma acentuada parcela de ocupação de suas CPUs. Algo que prejudica a execução de outras atividades que originalmente possam ser de sua exclusiva responsabilidade. Como comprovação de como a manipulação do fluxo de dados de uma rede por computadores de uso geral pode afetar seu desempenho em outras atividades, foi ilustrado por via de gráficos que relatam o percentual de consumo de CPU no processo de encaminhar dados intermediariamente entre outras máquinas. Tais testes simularam o *stress* sofrido por um servidor responsável também por encaminhamento de dados e mostrar que não somente as atividades de processamento de exclusiva responsabilidade do servidor acabam sendo prejudicadas, mas também a rede como um todo.

A primeira observação a ser feita no desenvolvimento do trabalho é que a ferramenta Click Modular Router é excelente em se tratando de desenvolvimento de novos conceitos de rede, aplicados ao tratamento da transmissão de dados. Entretanto, não possui performance competitiva com as demais ferramentas disponíveis no mercado, pois gera um excessivo consumo da CPU do servidor, provocando uma alta taxa de perda de pacotes e aumento da latência da rede.

No tocante à vazão dos modelos de transmissão de dados, o baseado em rajadas é superior aos demais, pois atinge uma maior vazão associada a um menor consumo da CPU dos servidores. Quanto as ferramentas de encaminhamento de dados testadas a que obteve o melhor desempenho, de forma geral, foi a manipulação nativa do Kernel linux, pois além apresentar a menor latência, também manteve a mesma praticamente constante, diferentemente das demais ferramentas. Seu desempenho é um pouco reduzido quando para pacotes de maior tamanho.

Portanto, conclui-se que com um eficiente método de classificação de pacotes, o modelo de rajadas associada com o encaminhamento via Kernel Linux se mostra como melhor opção na implementação das redes de *datacenter* centradas em servidores, alinhando uma maior vazão a um menor consumo da CPU.

6.3 Perspectivas

Com todos os dados coletados em laboratório percebe-se que a operação com o uso de rajadas em substituição as tradicionais modelos de comutação por pacotes e por circuitos tem seus benefícios quando associado a um sistema a de controle para monitorar a montagem, o envio das rajadas e a configuração do caminho de dados fim a fim. Tal observação pode ser confirmada uma vez que usando o *Click Modular Router* na montagem de redes baseadas em roteamento IP e encaminhamento por rajadas, Figura 5.6, 5.7 e 5.8, a última possui certa vantagem tanto na taxa de vazão, quanto no consumo de CPU mais reduzido, o que é consequência do fato do roteamento IP exigir uma análise maior do cabeçalho dos pacotes, enquanto que na rajada há apenas um repasse dos pacotes.

Entretanto nota-se que o consumo de recurso nos nós intermediários ainda é bem acentuado, devido ao grande fluxo. O uso chaves ópticas magnéticas, que possuem tempo de reposta que pode chegar à ordem de *nano* segundos, poderia em certos casos ser usadas para construir um ambiente parcial ou puramente óptico na transmissão das rajadas, substituindo quando possível os servidores quanto a tarefa de encaminhamento dos pacotes. Portanto, o uso de chaves ópticas, além de tornar a rede parcial ou totalmente transparente, contornaria o problema ligado ao consumo de CPU imposto aos comutadores, que por consequência está ligado a crescente porcentagem de perda de pacotes e aumento da latência, medidos em laboratório, já mostrados e comentados em seções anteriores deste trabalho.

6.4 Trabalhos Futuros

Como futuras propostas de trabalho, tem-se a possibilidade de se explorar a integração de novas tecnologias ao método de transmissão de dados por rajadas buscando aplicar as mesmas a ambientes característicos de *datacenters*. Para tanto, como possíveis tópicos, podem ser explorados o conceito de SDN, tecnologias como a API *OpenFlow*, *Light-Trail* e ferramentas como chaves magneto-ópticas e NetFPGAs, associadas as diversas modalidades de *Optical Burst Switch* (OBS), já citadas no Apêndice A2. Pode-se também explorar a modalidade de *Hybrid Optical Burst Switch* (OBS Hibrido), com o objetivo de analisar a melhor associação e aplicabilidade, buscando encontrar um modelo de rede flexível e adaptável às necessidades do fluxo de dados [23][24].

6.4.1 Hybrid Optical Burst Switch

Uma abordagem tratada como muito promissora dentro da comutação por rajadas ópticas é a comutação óptica de *Hybrid Optical Burst Switching (HOBS)* [24][35]. De forma geral a comutação híbrida combina ambas as vantagens de circuitos e paradigmas da comutação de pacotes a fim de aumentar a eficiência de utilização do link óptico, diminuir o número requerido de comprimentos de onda e reduzir a carga de tráfego processado de forma eletrônica pelos roteadores IP.

Nos nós de ingresso do HOBS tem-se uma abordagem de classificar o tráfego de entrada em pequenos fluxos (tráfego de melhor esforço) e grande fluxos. Pequenos fluxos são transportados utilizando OBS (através HOS-B módulos) e os grandes fluxos são transportados usando OCS (HOS-C através de módulos). Módulo HOS-B e HOS-C podem coexistir nos nós principais da arquitetura HOS, e competir pelos comprimentos de onda durante o processo de reserva de recursos.

Como uma das proposições para futuros trabalhos pretende-se estudar tal arquitetura de HOBS visto que embora a mesma seja mais complexa que a comutação por circuitos e por

rajadas, ao mesmo tempo se apresenta como uma técnica bastante flexível de uso do meio de transmissão.

6.4.2 *Openflow* e a integração com uma arquitetura hibrida

Um tecnologia que tem ganhado muita força recentemente é a de redes definidas por software (SDN), sendo o *Openflow* um de seus mais populares representantes. A tecnologia *OpenFlow* apoia-se nesse conceito de redes definidas por software, como uma interface de programação de aplicativos que definem o comportamento do encaminhamento de quadros na rede. Além disso, a tecnologia define um protocolo entre os comutadores e o controlador, que viabiliza o controle logicamente centralizado. Isto promove uma versatilidade no encaminhamento de quadros no plano de dados, controlados externamente por aplicações em execução nos sistemas operacionais de rede, os quais são executados nos controladores. O *OpenFlow* se apresenta como uma proposta de padrão de interface pragmática, com um protocolo definido entre plano de controle e plano de dados, permitindo a instalação de regras de encaminhamento nas tabelas de fluxos dos equipamentos. Estes fluxos são baseados em diversos parâmetros de protocolos de camadas distintas como Ethernet, IP, TCP e UDP, e em versões mais recentes do *OpenFlow*, inclui MPLS. Atualmente ele é a implementação mais difundida tanto no meio acadêmico, como nas implementações de equipamentos e sistemas comerciais de SDN [13][14].

Tomando por base a flexibilidade do *Openflow*, um possível trabalho seria a integração da tecnologia de SDN com a de redes ópticas OPS, OCS e OBS, mesclando a capacidade das redes ópticas com flexibilidade inerente do SDN, aliado a seu plano de controle e sinalização já bem desenvolvido.

Bibliografias

[1] Labovitz, Craig. *"Internet Traffic Evolution 2007 - 2011"*, Global Peering Forum, April, 6, 2011.

[2] K.G. Coffman and A.M. Odlyzko. *"Internet Growth: Is There a Moore?s Law For Data Traffic?"* Handbook of Massive Data Sets. Kluwer Academic Publishers, 2002.

[3] Gebert, S.; Pries, R.; Schlosser, D.; Heck, K. *"Internet Access Traffic Measurement and Analysis."* LECTURE NOTES IN COMPUTER SCIENCE, 7189; 29-42, Traffic monitoring and analysis. International workshop; 4th. Springer, Heidelberg, 2012, 12 pages. ISBN 3642285333

[4] Albert Greenberg, James Hamilton, David A. Maltz, Parveen Pate. *The Cost of a Cloud:Research Problems in Data Center Networks"*. ACM Sigcomm, 2009.

[5] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, I. Stoica *"A Cost Comparison of Data Center Network Architectures"*. 6 th International Conference CoNEXT, 2010.

[6] J. Peters, J. Davidson, M. Bathia, S. Kalidindi, S. Mukherjee. *Fundamentos de VoIP*. Editora Bookman, 2008. 392 p. ISBN.: 9788577801138

[7] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "*The Click Modular Router*," *Most*, no. Section 2, pp. 217–231.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *Technology*, vol. 1, no. 212.

[9] E. Kohler, "The Click Modular Router," *Electrical Engineering*, no. February, 2001.

[10] D. Guedes, VIEIRA, L. F. M., VIEIRA, M. A. M., H. Rodrigues, NUNES, R. V. *"Redes Definidas por Software: uma abordagem sistêmica para o desenvolvimento de pesquisas em Redes de Computadores."* SBRC 2012.

[11] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and S. Louis, "OpenFlow : Enabling Innovation in Campus Networks," *Internetworking Research And Experience*. ACM SIGCOMM, April 2008.

[12] V. Implemented and W. Protocol, "OpenFlow Switch Specification List of Figures," *Wire*, pp. 1–56, 2011. Disponível em: "http://archive.openflow.org/wp/learnmore/"

[13] O.N. F. White Paper, "Software-Defined Networking : The New Norm for Networks," April 2012.

[14] OpenFlow Specification v1.1.0. Disponível em: *"http://www.openflow.org/"*, acessado em 12 de março de 2013.

[15] Al-Fares, M., Loukissas, A. e Vahdat, A. *"A scalable, commodity data Center network architecture."* ACM *Sigcomm*, 2008, p. 63-74.

[16] A. Singla, C. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking Data Centers Randomly." 9th USENIX conference on Networked Systems Design and Implementation, 2012.

[17] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell : A Scalable and Fault-Tolerant Network Structure for Data Centers." ACM SIGCOMM, 2008.

[18] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, *"BCube : A High Performance , Server-centric Network Architecture for Modular Data Centers*." SIGCOMM, 2009.

[19] A. S. Tanenbaum, Computer Networks, Fourth Edition, Prentice Hall PTR, 2003.

[20] S. J. B. Yoo, H. Yokoyama, and Y. Horiuchi, "*Performance comparison of optical burst and circuit switched networks*" *OFC/NFOEC Technical Digest. Optical Fiber Communication Conference, 2005.*, p. 3 pp. Vol. 3, 2005.

[21] KUROSE, J. F.; ROSS, K.W. Redes de Computadores e a Internet: uma nova abordagem. Tradução de Arlete Simille Marques. São Paulo: Addison Wesley, 2003.

[22] Y. Chen, C. Qiao, X. Yu, and C. Science, "Optical Burst Switching (OBS): A New Area in Optical Networking." pp. 1–13, Journal of High Speed Networks, January 1999.

[23] M. Y. Sowailem, D. V Plant, and O. Liboiron-ladouceur, "Implementation of Optical Burst Switching in Data Centers," vol. 4, no. c, pp. 445–446, Photonics Conference (PHP), 2011 IEEE.

[24] H. H. Bazzaz, M. Tewari, G. Wang, G. Porter, T. S. E. Ng, D. G. Andersen, M. Kaminsky, M. A. Kozuch, and A. Vahdat, "Switching the Optical Divide : Fundamental Challenges for Hybrid Electrical / Optical Datacenter Networks." ACM Symposium on Cloud Computing, October 2011.

[25] Linux Network Administrators Guide.," *The Journal of infectious diseases*, vol. 207, no. 12, p. NP, Jun. 2013.

[26] M. Rio. *"A Map of the Networking Code in Linux Kernel 2.4.20"*, Research and Technological Development for a TransAtlantic Grid DataTag. no. March, pp. 1–41, 2004.

[27] B. D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd Edition, pg. 944. 2005, ed. O'Reilly Media. ISBN: 0596005652.

[28] Willmann, Paul. Rice University, *"A 10 Gigabit Programmable Network Interface Card - How NICs Work."*, Disponível em: http://www.ece.rice.edu/~willmann/past_enc.html, Acessado em 10 de julho de 2013.

[29] Manual IPtables – The netfilter.org "IPtables" project, disponível em: "*http://www.netfilter.org/projects/iptables/*", acessado em 25 de janeiro de 2013

[30] Manual Kernel Bridge Linux, disponível em: *"http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge*", acessado em 10 de abril de 2013.

[31] Manual Ebtables – Linux Ethernet bridge firewalling, disponível em: *"http://ebtables.sourceforge.net/*", acessado em 20 de janeiro de 2013

[32] Manual *OpenVswitch* disponível em: *"http://openvswitch.org/support/*", acessado em 21 de maio de 2013.

[33] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, *"OFLOPS: An Open Framework for OpenFlow Switch Evaluation."*, pp. 85-86, 13th International Conference, PAM 2012, Vienna, Austria, March 12-14th, 2012.

[34] R. E. Z. De Oliveira, P. P. P. Filho, M. R. N. Ribeiro, and M. Martinello, "Parâmetros balizadores para experimentos com comutadores OpenFlow : avaliação experimental baseada em medições de alta precisão .," pp. 13–16,SBrT, 2012.

[35] K. Vlachos and K. Ramantas, *"A non-competing hybrid optical burst switch architecture for QoS differentiation"* Optical Switching and Networking, vol. 5, no. 4, pp. 177–187, Oct. 2008.

[36] NAOUS, J.; GIBB, G.; BOLOUKI, S. Netfpga: reusable router architecture for experimental research.Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow, 2008. Disponível em: <http://portal.acm.org/citation.cfm?id=1397720>.

Apêndice 1: *FrameWork Oflops*

O Oflops é uma plataforma de testes idealizada e desenvolvida para testes de arquitetura OpenFlow. O principal foco da plataforma é fornecer um conjunto de ferramentas básicas para medição que permita aos desenvolvedores entender e quantificar as capacidades de um dispositivo OpenFlow e identificar possíveis gargalos, focando em aspectos do desempenho da aplicação OpenFlow.

Dentre as características da plataforma podemos citar:

- **Modularidade:** a plataforma é dividida em 2 partes. Existe o programa executável OFLOPS, que implementa a funcionalidade central da plataforma e há ainda um conjunto de bibliotecas dinamicamente carregadas que implementam as funcionalidades necessária para testes específicos;

- **Baixa Sobrecarga:** a plataforma é centrada na executação em múltiplos níveis de medições de alta precisão, para poder avaliar o desempenho *switch*. A fim de alcançar um nível de mínimo atraso nos pacotes, busca-se reduzir qualquer atraso no processamento, paralelizando o processamento;

- **Heterogeneidade:** A plataforma permite a integração com a geração de pacotes e mecanismos de captura. A plataforma pode rodar em PCs com multiplas interfaces de rede (NIC), assim como em sistemas equipados com placas NetFPGA [36]. Cada uma das plataforma oferece diferentes garantias de precisão e custos.

A tecnologia *OpenFlow* apoia-se no conceito de redes definidas por *software*, como uma interface de programação de aplicativos que definem o comportamento do encaminhamento de quadros na rede. Além disso, a tecnologia define um protocolo entre os comutadores e o controlador, que viabiliza o controle logicamente centralizado. Isto promove uma versatilidade no encaminhamento de quadros no plano de dados, controlados externamente por aplicações em execução nos sistemas operacionais de rede, os quais são executados nos controladores. O *OpenFlow* se apresenta como uma proposta de padrão de interface pragmática, com um protocolo definido entre plano de controle e plano de dados,

permitindo a instalação de regras de encaminhamento nas tabelas de fluxos dos equipamentos. Estes fluxos são baseados em diversos parâmetros de protocolos de camadas distintas como Ethernet, IP, TCP e UDP, e em versões mais recentes do *OpenFlow*, inclui MPLS. Atualmente ele é a implementação mais difundida tanto no meio acadêmico, como nas implementações de equipamentos e sistemas comerciais de SDN.

Como uma boa opção para se realizar testes de desempenho em redes SDN é o *framework* de medição *Oflops* [33], uma ferramenta que viabiliza o desenvolvimento rápido de testes de casos de uso para implementações *OpenFlow* tanto em *software* quanto em *hardware*. O *framework* conta com um gerador de quadros que funciona tanto em nível de *software* quanto de *hardware* que viabiliza medições versáteis e com baixo custo de operação. Um *PC* básico com Linux e uma placa *NetFPGA* Gigabits são os recursos mínimos que viabilizam medições de alta precisão na escala de microssegundos. Neste presente trabalho, buscou-se verificar aspectos funcionais básicos desse *framework*. A principal motivação é conhecer de maneira prática sua real capacidade e suas eventuais limitações. Isto foi feito de maneira complementar a avaliação comparativa entre a implementação *OpenFlow* padrão em *hardware* e em *software*.

Na sequencia tem-se um exemplo de configuração de teste com *Oflops* + NetFPGA usando o pacote *action delay* do *Oflops*.

```
#Example configuration file of Oflops

oflops: {
 control: {
control_dev = "eth0";
control_port = 6633;
snmp_addr = "200.137.64.212";
cpu_mib="1.3.6.1.2.1.25.3.3.1.2.768;1.3.6.1.2.1.25.3.3.1.2.769;1.3.6.1.2.1.25.3.3.1.2.770;1.3.6.1.2
5.3.3.1.2.771";
in_mib="1.3.6.1.2.1.2.2.1.11.6";
out_mib="1.3.6.1.2.1.2.2.1.17.6";
snmp_community = "public";
};

data = ({
 dev="nf2c0";
 port_num=1;
 in_snmp_mib="1.3.6.1.2.1.2.2.1.11.7";
 out_snmp_mib="1.3.6.1.2.1.2.2.1.17.7";
 type="nf2";
},{
 dev="nf2c1";
 port_num=2;
 in_snmp_mib="1.3.6.1.2.1.2.2.1.11.8";
 out_snmp_mib="1.3.6.1.2.1.2.2.1.17.8";
 type="nf2";
},{
 dev="nf2c2";
 port_num=3;
 in_snmp_mib="1.3.6.1.2.1.2.2.1.11.9";
 out_snmp_mib="1.3.6.1.2.1.2.2.1.17.9";
 type="nf2";
}, {
 dev="nf2c3";
 port_num=4;
 in_snmp_mib="1.3.6.1.2.1.2.2.1.11.10";
 out_snmp_mib="1.3.6.1.2.1.2.2.1.17.10";
 type="nf2";
});

traffic_generator = 3;
dump_control_channel=0;

module: ({
path="/root/oflops/example_modules/openflow_action_delay/.libs/libopenflow_action_delay.so";
param="data_rate=10 pkt_size=150 action=1/150,0/3 table=0 print=1";
});
};
```

Figura A1. 1 - Arquivo de configuração exemplo do Oflops + NetFPGA.

A1.1 NetFPGA

A NetFPGA, Figura A1.2, é uma plataforma de hardware reconfigurável de baixo custo, otimizado para redes de alta velocidade. A NetFPGA inclui recursos de lógica e memória associadas a interfaces Gigabit Ethernet possibilitando a para construção de roteador e/ou dispositivos de segurança. Sendo a manipulação dos dados completamente implementada hardware, o sistema pode suportar pacotes back-to-back em taxas que podem chegar a ordem de Gigabit e com uma latência de processamento medida em poucos ciclos de relógio.

O objetivo do desenvolvimento da plataforma NetFPGA é possibilitar a rápida prototipagem de hardware de alto desempenho para rede de computadores, sendo seu principal foco o ensino e pesquisa [36]. Uma NetFPGA é uma placa que é conectada ao barramento PCI ou PCI-Express de um computador pessoal. Existem atualmente dois modelos de NetFPGA. Um deles possui barramento PCI e 4 portas Gigabit Ethernet, já o outro utiliza um barramento PCI-Express e possui quatro portas SFPs (Small Form-factor Pluggable transceiver) com capacidade de 10Gbps. Os dois tipos de barramentos PCI são utilizados para programação da FPGA da placa e como interface de comunicação entre a mesma e o processador do computador. A placa também possui a facilidade de acesso direto à memória, DMA, tornando esta comunicação menos custosa do ponto de vista de processamento.

Há três projetos básicos desenvolvidos e disponíveis para NetFPGA: um interface de rede com 4 portas Ethernet, um switch Gigabit Ethernet e um roteador IPv4, todos implementados no padrão Gigabit Ethernet. Como de modo geral a maior parte dos projetos a serem desenvolvidos com a plataforma, pode ser baseada em extensões dos três projetos de disponíveis anteriores, acaba por torna a plataforma bem atrativa do ponto de vista da flexibilidade.

Como exemplo, o roteador IPv4 básico, implementado para Gigabit Ethernet, pode tratar o tráfego das 4 interfaces disponíveis em full-duplex, além de inclui o encaminhamento de pacotes em hardware, dois programas que permitem construir rotas e tabelas de roteamento, e uma linha de comandos de terminal e interface gráfica para gerência do roteador.

Figura A1. 2 - Placa NetFPGA de 1Gbps.

Embora a ferramenta apresente grandes benefícios no tocante ao desempenho e de desenvolvimento de hardware dedicado e otimizado, o trabalho de utilização da NetFPGA pode requerer um maior tempo de dedicação ao estudo mais aprofundado de sua linguagem de descrição de hardware, Verilog ou VHDL, mesmo já havendo a disponibilidade de exemplos de projetos disponibilizados pelos desenvolvedores e pesquisadores. Além disto, existe a necessidade de um computador onde a placa NetFPGA possa ser instalada, tornando o custo maior que o de roteadores implementados em software diretamente sobre estes mesmos PCs.

Apêndice 2: Formas de encaminhamento de *Burst Switching*

A2.1 Tell-And-Go (TAG)

Este é um esquema de reserva imediata. Em TAG, a pacote de controle é transmitido sobre um canal de controle seguido de uma rajada, por um canal de dados com zero ou offset desprezível. A rajada é retardada utilizando fibra de retardo (FDL) em cada um dos nós intermediários, enquanto o pacote de controle é processado nos mesmos nós. Se houver disponibilidade de recursos a reserva será bem sucedida, então a rajada ou *burst* é transmitida por via dos enlaces reservados, do contrário o *burst* de dados é descartado e uma confirmação negativa (NAK) é enviada para a fonte ou origem. O nó fonte envia um outro pacote de controle após transmitir toda a rajada a fim de libertar os recursos associados a sua transmissão no enlace. O inconveniente presente neste sistema é a disponibilidade de retardo óptico com o uso de fibra optica de atraso (FDL). O limitante no uso da FDL é o tempo de retenção fixo, uma vez que a mesma pode apenas manter os dados por um período fixo, em geral reduzido, e não podendo por consequencia acomodar os bursts de dados com tamanhos variádos. Além disso, outro inconveniente do TAG está no caso da perda do pacote de controle responsável por liberar os recursos reservados, resultando em desperdício de largura de banda e deixando recursos ociósos.

A2.2 Just-In-Time (JIT)

O modelo de arquitetura JIT, assim como o TAG, também se enquadra no modelo baseado em reserva imediata. Possuindo como característica a reserva de recursos nos nós, assim que o pacote de configuração é processado. De uma forma simplificada, o JIT atua da seguinte forma, os recursos necessários para a transmissão do *burst* são reservados imediatamente após a recepção pelo nó do pacote de configuração, caso os recursos não estejam disponíveis a mensagem de configuração é rejeitada e o *burst* que vem em seguida será descartado naquele nó. A idéia por trás da arquitetura JIT é estimar um intervalo de

tempo entre o pacote de configuração e o *burst* de forma que tal intervalo seja grande o suficiente para possibilitar o processamento do pacote de controle e a configuração de cada nó antes da chegada do *burst*, como pode-se observar pela Figura A2.1. A fonte de tráfego ou nó de entrada da rede transmite o burst ou rajada após um certo intervalo de tempo maior que o necessário para o processamento do pacote de configuração e a própria configuração dos nós intermédios. Caso recursos como determinado comprimento de onda não estejam disponíveis, a configuração é rejeitada e o *burst* que foi enviado após é descartado. A diferença entre o JIT e TAG está no fato de que no JIT o retardo da carga útil de dados, o burst ou rajada, por FDL para o processamento do pacote de controle em cada nó é eliminado, sendo substituido pela inserção de um intervalo de tempo entre o pacote de controle, ou configuração, e a rajada. A idéia da substituição de um atraso em cada nó por um intervalo de tempo grande no nó de ingresso acaba por simplificar os nós de núcleo do JIT quando comparados com os do modelo TAG. A partir do momento imediatamente após a reserva dos recursos com o recebimento do pacote de configuração até o momento em que o início do burst chegue ao nó todos os recusos reservados encontram-se ociosos. Isto está ligado a diferença temporal entre o pacote de controle e a carga útil. Por via de indicar o termino da reserva pode ser posto no fim da rajada um indicador. Na Figura A2.1 encontra-se ilustrado de forma simplificada a operação do JIT, onde o nó denominado *origem* envia uma mensagem de configuração ao próximo nó, o *switch 1*, e só após um intervalo de tempo *Offset inicial* é enviado o *burst*. Já em *switch 1*, observa-se o tempo gasto para o processamento da mensagem de configuração, juntamente com o tempo necessário para a reserva de recursos e configuração (t_{OXC}), que no caso da figura é o próprio comprimento de onda, sendo simultaneamente encaminhada ao nó seguinte a mensagem de configuração, restando-se ainda um intervalo de tempo que será usado na repetição do mesmo processo de reserva e configuração nos nós seguintes até o nó *destino*.

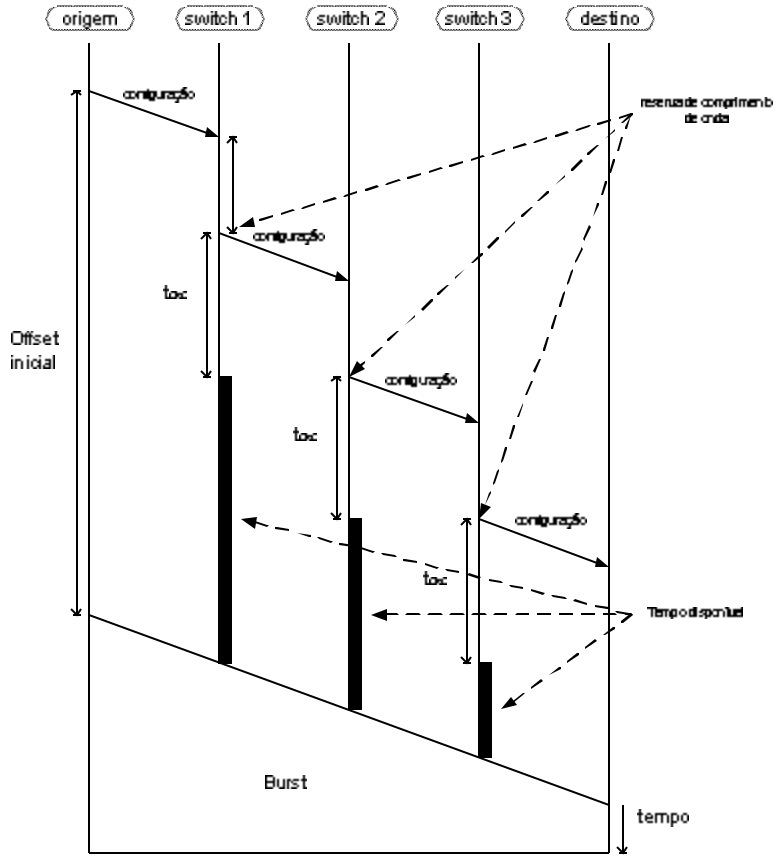

Figura A2. 1 - Modelo de reserva Just-In-Time

A2.3 Just-Enough-Time (JET) e Horizon

Tanto o JET quanto o Horizon são exemplos de arquiteturas baseadas em reserva por atraso. Em ambas o tamanho do *burst* de dados é decidido antes de o pacote de controle referente a rajada ser enviado pela fonte. O deslocamento, ou intervalo de tempo, entre o pacote de controle e a carga também é calculado com base no número de saltos existênte entre a origem e o destino. O intervalo de tempo entre o pacote de controle e o *burst* de dados é composto por uma parcela destinada ao processamento da mensagem de controle, com a verificação da disponibilidade ou não de recursos a rajada precedida pelo pacote, parcela reservada ao tempo gasto na configuração do nó e ainda uma parcela ociosa do ponto de vista da rajada que será recebida mas não do ponto de vista do próprio nó, já que o mesmo pode estar usando esse mesmo tempo para encaminhar outras rajadas provenientes de outros nós.

Como pode-se observar na Figura A2.2, na arquitetura baseada em reserva atrasada a configuração do nó só se realiza instantes antes da chegada do primeiro bit de dados do *burst*. Outro ponto a ser observado é que esse tempo dito ocioso não é um intervalo fixo, mas varia segundo o número de nós que compõe o caminho entre origen e destino. Em cada nó, se há recursos disponíveis e o intervalo de tempo entre o quadro de controle e o *burst* se enquadram com a disponibilidade de *slots* de tempo do nó, então a mensagem é aceita e as intruções para a configuração são armazenadas, aguardando até o momento de ser aplicada. Caso não haja recursos disponíveis o pacote de controle é desconsiderado e o *burst* é descartado no momento em que atingir o nó. A idéia por trás da reserva atrasada é que o processo de configuração do elemento de comutação é postergada o máximo possível, reduzindo a ociosidade do nó e dos recursos do enlace, possibilitando que pelo nó possam ser transmitidos outros *bursts* no intervalo entre um mensagem de configuração e seu respectivo *burst*, o que gera uma maior capacidade de utilização do meio, embora associada a uma maior complexidade do processo de agendamento da comutação quando comparado a outras arquiteturas.

A diferença entre as arquiteturas JET e o Horizon se encontra basicamente na forma como são agendados os *bursts*. No Horizon é definido um tempo chamado de tempo de horizonte, de onde vem o próprio nome da arquitetura, tempo este que corresponde ao tempo necessário para que o último *burst* aceito seja transmitido, o que engloba tempo de configuração do nó e o tempo de duração da rajada, logo se uma mensagem de configuração para uma nova rajada apresente o tempo para o início maior que o do último *burst* aceito, este também será aceito, já se o tempo for menor e exista um intervalo de tempo ocioso entre as mensagens de configuração e os demais bursts aceitos o pacote de configuração do novo burst será rejeitado e o burst descartado. O JET em oposição ao *Horizon*, numa tentativa de usar os vazios criados pelas mensagens de configuração de outras rajadas, é capáz de agendar configurações de rajadas entre duas já agendadas, mesmo a mensagem de configuração da última tenha chegando ao nó após as mensagens dos outros *bursts*. Portanto o JET é mais eficiente que o *Horizon* em se tratando da ocupação do meio, entretanto possui um agendamento mais complexo.

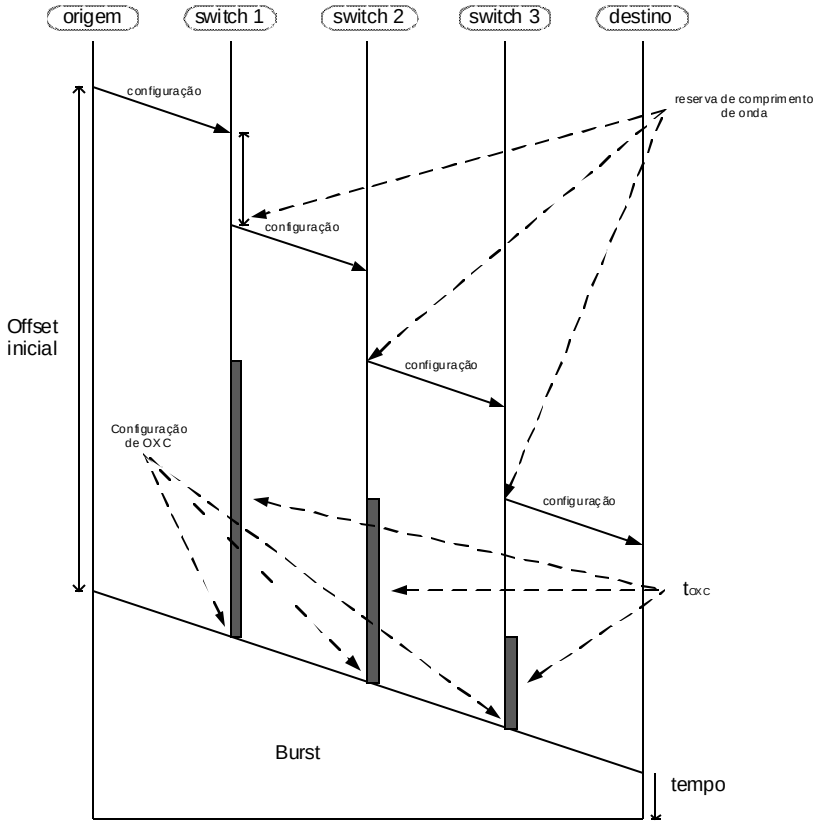

Figura A2. 2 - Modelo de Reserva Just-Enough-Time